Technical Report UMIACS-TR-2002-12, Institute for Advanced Computer Studies,
University of Maryland at College Park, February 2002.
Also Computer Science Technical Report CS-TR-4330.

# System Synthesis for Polymorphous Computing Architectures

Sumit Lohani and Shuvra S. Bhattachrayya

Department of Electrical and Computer Engineering, and

Institute for Advanced Computer Studies

University of Maryland, College Park

{slohani,ssb}@eng.umd.edu

## Abstract

In general, polymorphous computing architectures are architectures that can be dynamically customized to various applications. This report is concerned with metrics, formulations, and algorithms for systematically synthesizing architectural configurations and software for such architectures, particularly in the domain of digital signal processing (DSP).

In polymorphous system synthesis for DSP, one central aspect is managing trade-offs between latency and throughput, which are critical metrics for DSP applications. In Sections 1-4 of the report, we develop a model for latency that is more appropriate than conventional models of latency for DSP system synthesis, and that takes into account central issues related to transient-state time, and we develop precise relationships between latency and throughput using this framework. Schedule post-processing strategies based on simulation and retiming that reduce the latency and transient for a given throughput constraint are then presented, and their efficacy is substantiated with experimental results on a number of practical DSP benchmarks. Also, a streamlined approach based on a graph-theoretic framework is suggested that leads to much faster execution of the proposed schedule post-processing techniques.

Sections 5-6 of the report then deal with the problem of efficient mapping of an application with stochastic execution times to a polymorphous computing architecture in accordance with the time-varying performance requirements for several metrics, which may include even non-trivial metrics such as the coupled latency/throughput metrics that are addressed earlier in the report. A comprehensive model for system synthesis in this context is developed; results are developed on the complexity of system synthesis under this model; and preliminary algorithms that address the synthesis problem are presented, and evaluated experimentally.

# Report Documentation Page

*Form Approved*
*OMB No. 0704-0188*

| | | |
|---|---|---|
| 1. REPORT DATE<br>**FEB 2002** | 2. REPORT TYPE | 3. DATES COVERED<br>**00-00-2002 to 00-00-2002** |
| 4. TITLE AND SUBTITLE<br>**System Synthesis for Polymorphous Computing Architectures** | | 5a. CONTRACT NUMBER<br>5b. GRANT NUMBER<br>5c. PROGRAM ELEMENT NUMBER |
| 6. AUTHOR(S) | | 5d. PROJECT NUMBER<br>5e. TASK NUMBER<br>5f. WORK UNIT NUMBER |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>**University of Maryland,Department of Electrical and Computer Engineering,Institute for Advanced Computer Studies,College Park,MD,20742** | | 8. PERFORMING ORGANIZATION REPORT NUMBER |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) | | 10. SPONSOR/MONITOR’S ACRONYM(S)<br>11. SPONSOR/MONITOR’S REPORT NUMBER(S) |
| 12. DISTRIBUTION/AVAILABILITY STATEMENT<br>**Approved for public release; distribution unlimited** | | |
| 13. SUPPLEMENTARY NOTES | | |
| 14. ABSTRACT | | |
| 15. SUBJECT TERMS | | |

| 16. SECURITY CLASSIFICATION OF: | | | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| a. REPORT | b. ABSTRACT | c. THIS PAGE | | | |
| **unclassified** | **unclassified** | **unclassified** | | **103** | |

## 1. Introduction

For an application with actors that have deterministic execution times, and that is executing in a self-timed manner on a multiprocessor system, the beginning of execution is some finite transient that settles down to a periodic pattern eventually [2, 37]. The period of this pattern depends on several factors, including the schedule, the execution times of computational tasks, and the delays in the critical cycle of the dataflow graph [23, 30]. This phenomenon has been discussed in many references (e.g., see [2, 33]).

In many Digital Signal Processing (DSP) applications, for a given periodic input sequence, only a periodic output sequence is useful, and we can withstand only a limited delay in generating the periodic output sequence from the arrival of the input. It is often desirable to have this input-output delay (the time that elapses between arrival of the first input sample and generation of the first sample in the periodic output sequence) as small as possible subject to a given sample-rate. If one has an output buffer where one can store output values and deliver them as needed (e.g., to maintain periodicity), the problem becomes one of a 3-way relationship between the throughput of the system, the output buffer size $B$ and the input-output delay. We will define this “delay” later as the periodic-output latency in this report, and explore the associated 3-variable relationship in greater depth with techniques for reducing periodic-output latency.

## 1.1. Definitions and notation

In the area of digital signal processing (DSP), dataflow is widely recognized as a natural model for specifying DSP applications. In dataflow, a program is represented as a directed graph, called a *dataflow graph*, in which vertices, called *actors*, represent computations and edges represent FIFO channels, (also called *buffers*). These channels queue data values, in the form of *tokens*, which are passed from the output of one actor to the input of another. When an actor is executed (fired), it consumes a certain number of tokens from its inputs, and produces a certain number of tokens at its outputs. The token input to an actor at an input edge stays there until the actor has finished execution and it is only at the end of execution that the actor produces tokens onto the output edges and consumes tokens from the input edges. The edges in the dataflow graph may contain *initial tokens*, which we also refer to as *delays*. Edges with delays can be interpreted as data dependencies across iterations of the graph. If there is a directed edge from an actor $v_1$ to an actor $v_2$ in the dataflow graph, then we say that $v_1$ is an *immediate predecessor* of $v_2$, and $v_2$ is an *immediate successor* of $v_1$. Also, for a directed edge from actor $v_1$ to $v_2$, the actor $v_1$ is called the *source* and $v_2$ the *sink* of that edge. The source and sink of an edge $e$ are denoted by $\mathrm{source}(e)$ and $\mathrm{sink}(e)$, respectively.

In this report, the set of all real numbers is denoted by $\Re$, and $\Re^n$ denotes a set whose elements are all $n$-dimensional sets of real numbers. The maximum of a finite set $S$ of real numbers is denoted by $max(S)$. The maximum of the empty set, $max(\Phi)$, is defined to be zero.

Assuming that a space $S$ has a distance function that defines the distance between any two points in space $S$, *triangle inequality* is said to hold in $S$ when the distance between any two points $a \in S$ and $b \in S$ is less than or equal to the sum of the distance between $a$ and any point $c \in S$ and the distance between $c$ and $b$. That is, in a space $S$ that satisfies triangle inequality, for any three points $a, b, c \in S$,

$$\text{distance}(a, b) \leq \text{distance}(a, c) + \text{distance}(c, b), \tag{1}$$

where $\text{distance}(x, y)$ represents the *distance function* that gives the shortest distance between any two points $x, y \in S$.

A space is called a *metric space* if the distance function is non-negative, symmetric and satisfies the triangle inequality. That is, a space $S$ is a metric space if for any three points $a, b, c \in S$,

$$\text{distance}(a, b) \geq 0, \tag{2}$$

$$\text{disance}(a, b) = \text{distance}(b, a), \text{ and} \tag{3}$$

$$\text{distance}(a, b) \leq \text{distance}(a, c) + \text{distance}(c, b), \tag{4}$$

where $\text{distance}(x, y)$ represents the distance function that gives the shortest distance between any two points $x, y \in S$.

**Example 1:** Figure 1 shows an example of a dataflow graph. Numbers beside edges indicate non-zero delays. There exists a delay or initial token of 1 between actor C and actor D. ■

Dataflow models are a very useful specification mechanism for signal processing systems since they capture the intuitive expressivity of block diagrams, flow charts and signal flow descriptions.

**Self-timed execution:**

In a self-timed schedule, processor assignment and actor ordering are performed at compile time, but run-time synchronization is used to determine actor firing times: a self-timed schedule executes by firing each actor invocation $A$ as soon as it can be determined via synchronization that the actor invocations on which $A$ is dependent have all completed execution. Conceptually, the processor sending data writes data into a FIFO buffer, and blocks when the buffer is full; the receiver on the other hand blocks when the buffer it reads from is empty. Thus, the flow control is performed at run-time.

It has been shown that eventually any self-timed execution settles down into

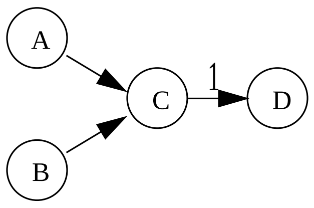


Figure 1. An example of a dataflow graph.

a periodic repeating pattern provided that contention of shared resources is resolved deterministically [4]. This repeating pattern can be exponential in the number of delays that are present in the critical paths of the schedule [37]. While checking for periodicity in the execution pattern, we consider the time for which any actor is executing on a processor as well as the delay distribution in the application graph at that time instant. We define the *transient-interval* or simply the *transient* in the execution as the time from the beginning of the execution to the first time when a periodic execution pattern is observed.

**Example 2:** Figure 2 shows an application graph that has 8 actors and the mapping of these actors onto five different processors. Figure 3 represents the self-timed exe-

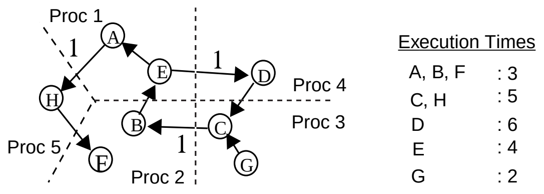


Figure 2. An example application graph.

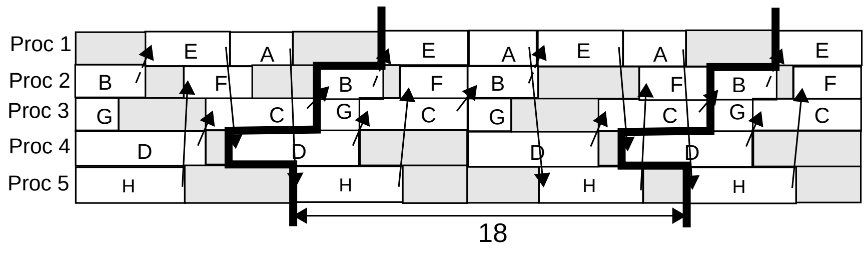


Figure 3. Self-timed execution.

cution pattern when the application graph in Figure 2 is executed in a self-timed manner. Two iterations of the schedule are carried out in 18 time units when synchronization costs are ignored, which gives an iteration period of 9. When the shown self-timed schedule settles down, it spans a period repeating period of 18 time units. One can notice that at time instant 11, when C finishes execution, the transient ends as at that instant the whole application has executed one iteration and after that there is a periodic pattern to follow. ■

The evolution of a self-timed implementation can be modeled by Sriram's *Inter Processor Communication (IPC) graph* model [36]. Given an application graph and an associated self-timed schedule, the IPC graph, denoted $G_{ipc}$, is constructed by instantiating a vertex for each application graph actor, connecting an edge from each actor to the actor that succeeds it on the same processor, and adding an edge that has unit delay from the last actor on each processor to the first actor on the same processor. Also, for each application graph edge $(x, y)$ that connects actors that execute on different processors, an *inter-processor edge* is instantiated in $G_{ipc}$ from $x$ to $y$. A sample application graph and a self-timed schedule are illustrated in Figure 4, and the corresponding IPC graph is illustrated in Figure 5.

IPC costs (estimated transmission latencies through the multiprocessor network) can be incorporated into the IPC graph model by explicitly including *communication* (*send* and *receive*) *actors*, and setting the execution times of these actors to equal the associated IPC costs.

It is well known that in the ideal case of unlimited bus bandwidth [30] and deterministic execution times, the average iteration period for the As-Soon-As-Possible (ASAP) execution of an IPC graph is given by the *maximum cycle mean* (*MCM*) of $G_{ipc}$, which is defined by

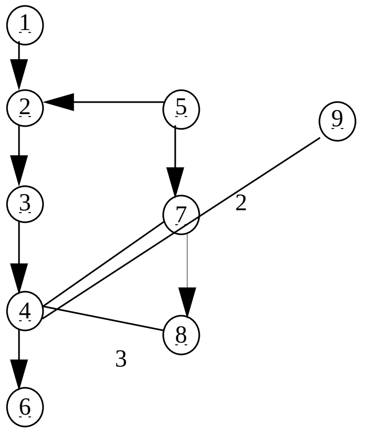
, (5)

$$MCM(G_{ipc}) = \max_{\text{cycle } C \text{ in } G_{ipc}} \left\{ \frac{\sum_{v \in C} exec(v)}{delay(C)} \right\}$$

where $exec(v)$ denotes the execution time of an actor $v$, and $delay(C)$ denotes the sum of tokens on all the edges of cycle $C$.

## 1.2. Latency in literature

In the engineering community itself, latency is defined in several ways and



Self-Timed Schedule

Proc 1: (1, 2, 3, 4, 6)

Proc 2: (5, 7, 8)

Proc 3: (9)

Figure 4. An example of an application graph and an associated self-timed schedule. The numbers on the edges $(6, 8)$ and $(6, 9)$ denote nonzero delays.

there is no universally accepted definitions of latency. Even in the embedded system design community, there are several different definitions of latency and people use the one that best suits the context. Some of the commonly used definitions of latency are the following.

- For dataflow graphs and related models, latency has usually been defined as the *response time* of the system to an input [18, 22, 35, 38]. Response time has been used in a general sense to refer to the time difference between the arrival time

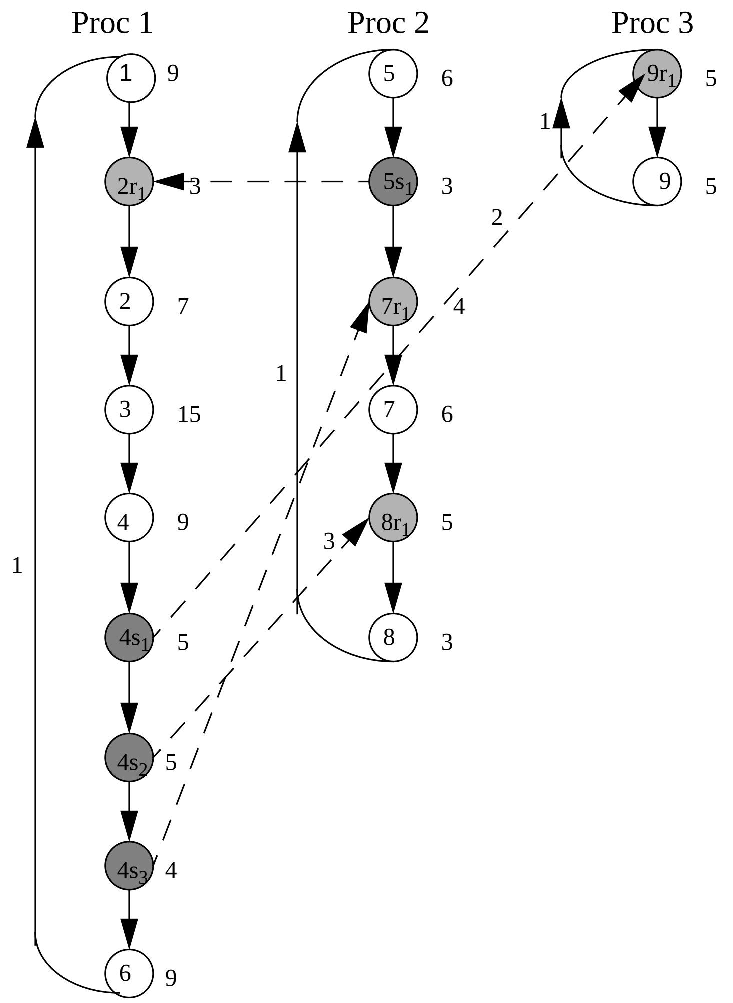


Figure 5. IPC graph constructed from application graph of Figure 4. Num bers besides communication actors denote communication costs.

of an input and the production time of the corresponding output. So, the latency is the time required for the first invocation of the input to influence the output, and thus the latency corresponds to the critical path in the dataflow implementation to the first output invocation that is influenced by the input. This interpretation of the latency as the critical path is widely used in VLSI signal processing [25, 37]. Henceforth, we will refer to this definition of latency as *response-time latency.*

• Latency according to the previous definition can be variable and data-dependent. So, latency has also been defined in the literature as the maximum of the time required for any invocation of the input to influence the output [31].

• In the context of computer networks, latency corresponds to how long it takes a message to travel from one end of a network to the other [31]. For most connections, a ping time of 100ms or lower is considered “low” latency; 100-500ms would be considered “moderate” latency; and 500-1000 ms would be “high” latency.

• Latency has also been defined to be the first time at which the sink vertex fires for the first time in the schedule, assuming the execution of the schedule started at time zero. Using this definition of latency, nested-schedules have a lower latency than flat, “single appearance” schedules [6].

• Latency is also defined as the time required to process a single data-set [11]. Note that this definition is different from the first definition because there may be synthetic delays present in the system.

• People have also proposed variants of latency as more suitable performance indices for specific classes of applications. One example is the concept of *control latency*, as proposed in [21] for task assignment and scheduling problems of multiprocessor real-time control systems. This control latency is defined as a weighted sum of feedback, command and monitoring latencies, which are defined as follows in [21]. Feedback latency is the time required for a sensor activation, control computation, and corresponding actuator manipulation. The time from the receipt of the command from the operator or host computer to the corresponding actuation is defined as *command latency*. *Monitoring latency* is the time required to pre-process the sensed data and then report it to host computer.

In all the above cases, irrespective of the exact definition of latency, a “low” value of latency is desirable and in many cases a “low” value of latency is critical.

For synchronous DSP system implementation, the notion of input-output timing present in the response-time latency is often adequate. However, for self-timed multiprocessor systems with DSP applications, which are becoming increasingly important due to problems with global clock distribution and other scaling issues and also due to increasing levels of dynamics in DSP applications, equating latency with response time is not as meaningful as a metric of latency that explicitly takes the periodicity of the output pattern into account. A more appropriate definition of latency for a periodic input signal could be stated intuitively as “The time difference between the appearance of the first periodic valid output instance and the application of the first input instance assuming an output buffer of given size”. Mathematically, this can be explained in the following way. Suppose that a given system is to satisfy a minimum throughput requirement $tr_{min}$. Suppose also that the associated iteration period is represented by $T$ where $T = 1/(tr_{min})$; the output buffer is represented by $\beta$; and the size of $\beta$ is represented by $B$. By definition, $\beta$ must satisfy the following three conditions.

1. The buffer $\beta$ can hold at most $B$ output samples at any time.
2. Once stored in the buffer, an output sample can be delivered at will.
3. An output sample can not be stored in the buffer before it is generated.

Let $x_1, x_2, \ldots, x_n, \ldots$ be an infinite sequence representing the arrival times of input samples 1, 2,.... $n$,.... and $y_1, y_2, \ldots, y_n, \ldots$ be the production times of the corresponding output samples. Define $\text{contains}(t, n)$ to be a binary-valued function that returns 1 if buffer $\beta$ holds the value of output sample $n$ at time instant $t$, and 0 otherwise.

Also, for a given iteration period $T$ and buffer size $B$, we define a valid output remapping (VOR) to be a sequence $y'_1, y'_2, \ldots, y'_n, \ldots$ that satisfies the following four conditions.

$$y'_n \geq y_n \text{ for all } n. \tag{6}$$

$$\text{If } y'_n \neq y_n\text{, then } \text{contains}(t, n) = \begin{cases} 1 \text{ if } \; y_n \leq t < y'_n \\ 0 \text{ otherwise} \end{cases} . \tag{7}$$

$$\text{For all } n \text{ such that } y'_n \geq L\text{, we have } (y'_{n+1} - y'_n) = T. \tag{8}$$

For all $t \geq 0$, $\sum_{k \varepsilon S_t} \text{contains}(t, k) \leq B$, where $S_t = \{n | \text{y}_n \varepsilon t\}$ and $n = \{1, 2, 3, \ldots\}$. That is, at every time instant, the buffer population is bounded by $B$. (9)

The minimum value of $L$ over all valid output mappings $\{y'_n\}$ is defined as the periodic-output latency of the system. One can check if some given value of $L$ satisfies all the above listed conditions [(6), (7), (8), (9)] and hence check if a VOR associated with that value of $L$ exists, in one pass over all output samples. So, to compute the minimum value of $L$, one can start with the value of $L$ as zero and check for its feasibility. This value of $L$ can be incremented until it becomes feasible i.e. there exists a VOR associated with that value of $L$. Thus, The VOR that gives minimum $L$ can be found with reasonable complexity using this approach. This approach is an *off-line* method for *periodic-output latency computation* and is described in detail later in this section. This approach for computing minimum $L$ can be streamlined using a binary search on the range of all possible values of $L$. In the streamlined approach that uses binary search, the range of the possible minimum $L$ values is halved repetitively on the basis of the feasibility of the average $L$ value in the range of possible minimum $L$ values. This streamlined approach computes minimum $L$ value more efficiently than the earlier approach, which computes minimum $L$ value by checking the feasibility of $L$ values in an incremental fashion. The problem addressed in this report is finding a schedule for an application, for a given $B$, such that the periodic-output latency $L$ is minimum, and the throughput is no less than $1/T$, when the application is executed in a self-timed manner according to that schedule on the given multiprocessor system. Henceforth, we will address this as *TBL problem.* Note that the individual processors in the multiprocessing environment can be arbitrary processing components, such as DSP processors, FPGAs or microcontrollers.

When defined in this way, periodic-output latency depends upon the output buffer size $B$ and requires the production times of output data to eventually become periodic. Until now, however we have not considered the fact that there may be initial tokens present in a schedule (e.g., due to retiming and related transformations), for the given application. For a more general framework that includes initial tokens, we have to distinguish between output values that are dependent on the input samples to the application and the ones that are completely independent of the input samples (i.e., the ones that are dependent only on initial tokens). We define the input samples to the application to be *valid input values*. Also, the output of an actor is a *valid output value* if and only if at least one of the input values to that actor is valid. Due to the presence of the initial tokens, some number of initial output samples may be invalid output values. So, in case there are initial tokens present in the schedule, the definition of periodic-output latency needs to be modified such that only valid output samples are considered.

The formulation stated above also implies that an increase in the number of initial tokens can lead to an increase in the above defined periodic-output latency. This introduces one more parameter in the relationship between periodic-output latency $L$, buffer size $B$, and iteration period $T$, but we will mainly concentrate on the relationship between $L$, $T$ and $B$, since we are assuming that the schedule, and hence the delay distribution, is given a-priori. From now on, in this report, the word “latency” has been used to describe the periodic-output latency, unless otherwise stated. Figure 6 illustrates a system with no output buffer. Figure 7 illustrates the same system when an output buffer of appropriate size is present and helps understand the relationship between periodic-output latency, buffering and throughput($=1/T$) of the system. One can observe that the buffer helps achieve lower periodic-output latency by storing the output samples temporarily and delivering them such that the associated VOR has a lower periodic-output latency.

Since a period of the execution pattern may contain more than one cycle of dataflow graph, in general, samples of the output sequence need not be equidistant. *Jitter* is a measure of unequal distances in the output sequence. In the literature, jitter has been defined in several ways [28, 39]. This is one attribute whose relationship with the other parameters described above can be monitored to get more insight

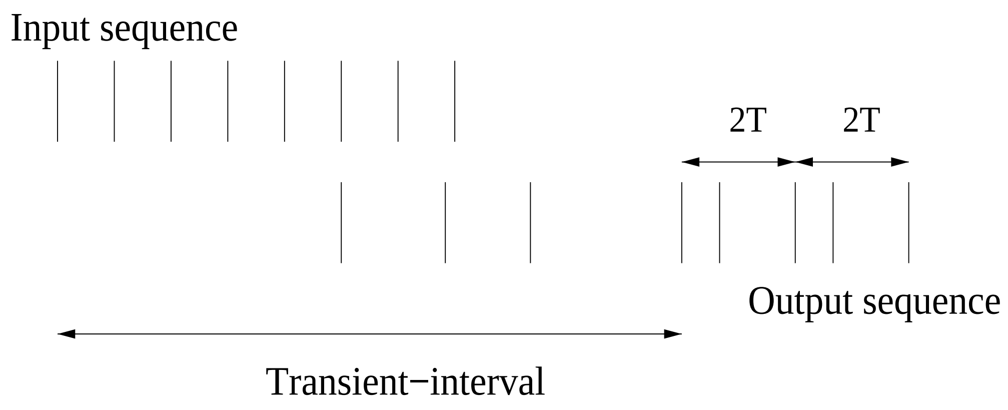


Figure 6. Illustration for a system with no output buffer.

into the concepts introduced above, but is out of the scope of this report. One may define jitter as the difference between the mean distance between output samples and distance between two consecutive output samples. We are designing for zero jitter, which is a reasonable goal if the jitter tolerance of an application is unknown or not specified to the designer/compiler. Exploring trade-offs associated with non-zero jitter allowance would be a useful direction for future work.

Intuitively, throughput would increase as periodic-output latency increases for a fixed buffer size, as larger periodic-output latency gives us more leeway in choosing the time instants when we can output values, which may correspond to improved throughput. An increase in buffer size leads to equal or improved achievable throughput. Also, for a fixed throughput value, an increase in buffer size may compensate for decrease in periodic-output latency in some cases. Note that even for a fixed output buffer one can find a set of (periodic-output latency, throughput) value pairs, as the execution can be done only for finite length of time. We are assuming that one has execution time estimates or a trace of the execution of the application.

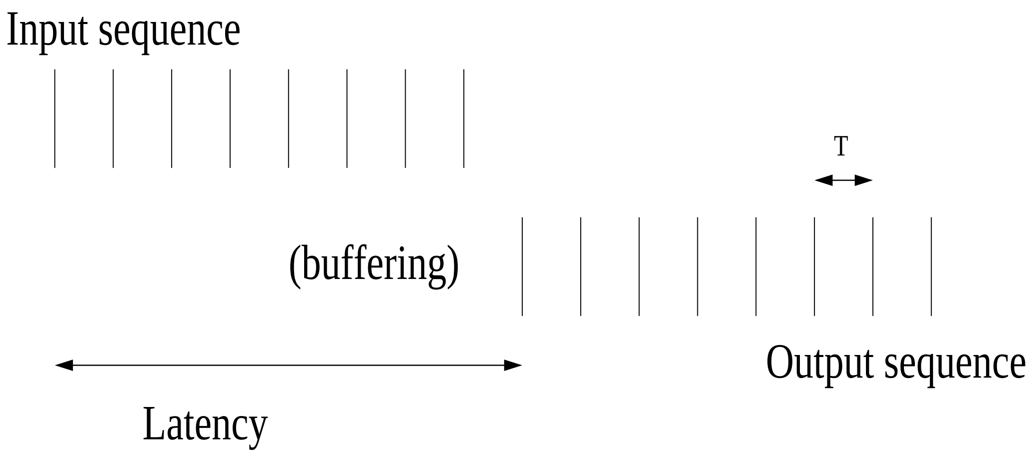


Figure 7. Illustration for latency, throughput and output buffer.

The relationship between buffer size, periodic-output latency and throughput can be calculated using an *off-line throughput-latency computation* scheme, which after fixing the throughput and buffer size, gives the periodic-output latency for the system from the execution record of the graph. This off-line throughput-latency computation scheme is as follows.

The data for off-line computation are the time instances of the output data samples for the periodic input. For a given buffer size and throughput, one can determine the periodic-output latency using the off-line method, which can be implemented by first starting with the lowest periodic-output latency value possible and seeing if that periodic-output latency is satisfiable using the given buffer size and throughput. If not, the periodic-output latency value is increased and checked for satisfiability again, until we find a periodic-output latency value that is satisfiable. We call this an off-line method since we have already simulated the execution of the system or have obtained a trace from an actual execution and are computing the minimum satisfiable periodic-output latency only after we have all of the relevant data.

One can observe that in the case of deterministic execution times, for a buffer size of zero, the periodic-output latency is approximately equal to the time taken for execution to settle to a periodic pattern, if the output at that instant is valid. Therefore it is helpful to have a schedule post-processing strategy to reduce the *transient* in the execution. One should note that the transient in a self-timed execution is not same as the periodic-output latency as illustrated in Figure 6 and Figure 7. Periodic-output latency is defined for a given output buffer unlike the case of the transient, where there is no output buffer.

## 1.3. Overview

This report presents two schedule post-processing strategies, the first one of which is to reduce the transient in execution pattern. The other schedule post-processing scheme, based on retiming, can be used to reduce latency for a given throughput constraint, that is, it gives us an approach to solve the TBL problem. The schemes proposed are discussed in conjunction with an initial scheduling scheme in the later sections of this report to give a complete approach to throughput-constrained latency minimization or throughput-constrained transient minimization. This approach is divided into two phases. The first phase consists of scheduling the graph using some known throughput-constrained or throughput minimizing scheduling technique such that the schedule satisfies the throughput constraint. One of the proposed schedule post-processing techniques constitutes the second phase. So one can use the appropriate schedule post-processing technique in phase 2 of the algorithm depending on whether the aim is to reduce the transient interval or to reduce the periodic-output latency i.e., the TBL problem. This report also explores the relationship between the output buffer size and the periodic-output latency to a limited extent. At the end, a new approach is suggested that leads to much faster execution of the proposed schedule post-processing techniques. This approach utilizes a known result relating the starting time of $m$th invocation of an actor to the longest $m$-delay path to that actor and finds out the starting time of any invocation of an actor using an efficient graph-theoretic framework.

The next section briefly describes the work done in the areas related to this report. Section 3 describes our two-phased approach consisting of a standard scheduling scheme for phase 1 and the proposed schedule post-processing schemes for phase 2. Experimental results for the schedule post-processing strategies are reported. Section 4 presents our streamlined approach to optimizing periodic-output latency for deterministic contention-free systems. This is a graph-theoretic approach that does not require event-based simulation and is based on mathematical properties of graphs and Petri-net theory [30], which leads to faster execution of the simulation-based, two-phased approach described in Section 3.

*Polymorphic* is defined as having, taking, or passing through many different forms or stages. In general, a *polymorphous computing architecture* (PCA) refers to any computing hardware that is modifiable. Commonly used polymorphous computing architectures, such as the Raw architecture [40], implement only a minimal set of mechanisms in hardware so as to allow the compiler to customize the hardware to different applications. Various high-level attributes of these architectures can be varied, such as inter-processor message routing, caching policies, scheduling policies, processor voltages, resource allocation to computing units, and architectural support for synchronization during inter-processor communication. This reconfigurability of PCA may help satisfy varying performance requirements that might not be possible if the architecture is fixed. This report also deals with the problem of mapping an application with stochastic execution times onto a polymorphous computing architecture in accordance with dynamically-varying performance requirements. In such a scenario, the performance requirements may change while the application is still executing. This problem is commonly encountered in various defense related applications. One example of this is a missile system which operates under varying performance requirements. A comprehensive model is suggested using which one would be able to deal with the problem of finding efficient mappings for continuously changing performance requirements. The motivation is to provide a model that is able to handle performance requirements of even non-trivial metrics such as periodic-output latency as defined in the report. The approach suggested in the report is quite general in nature and can handle diverse applications for a variety of metrics.

The PCA software synthesis problem described above is formulated and the motivation for our model to solve the problem and an introduction to it is given in Section 5. Section 6 describes the model in detail. Section 7 summarizes the results and discusses the efficacy of the proposed techniques and the model.

# 2. Background

Many scheduling techniques can be found in the literature to reduce the schedule makespan for a dataflow graph or to maximize the throughput of an iterative dataflow graph [5, 7, 41, 10]. Bokhari's algorithm [7] is an optimal algorithm for mapping a chain structured task graph onto a linear chain of processors. On the other hand, the throughput maximization scheme by Banerjee et al. [5] tries to compute a schedule for best throughput by identifying the problem as a combination of optimal task/processor assignment to pipeline stages as well as a scheduling problem. The scheduling problem was solved by them in two phases. In the first phase, a trade-off between clustering and parallelism was found using iterative scheduling techniques, and in the next phase the coarse solution was optimized through iterative refinement techniques.

Alternative scheduling problems, such as finding a schedule that satisfies a throughput constraint and has minimum response time and vice-versa, have also been studied [13, 19, 3]. Choudhry et al. [13] have solved this problem in a relatively constrained framework, where the "execution signature" or "response time function" for each node of the dataflow graph, is given. The execution signature is defined as the performance speedup as a function of the number of processors employed. Their approach is to decompose any serial-parallel graph into a series of serial and parallel components and then find the optimal assignment of processors to different tasks in the task graph so that the response time is minimum for a given throughput constraint and vice-versa. Aiken and Nicolau [1] find “pattern” in the execution history to find a schedule that maximizes throughput for a loop. Their “pattern” in the execution history can be related to the periodicity in self-timed execution.

Solving the TBL problem is different from the earlier work in the sense that this periodic-output latency relates to the settling time of the pattern, which is crucial in many DSP applications. For the transient-reduction problem, it is computationally intensive to find the settling time of a given schedule and hence a minimum transient schedule. There are not many known properties of a graph that can be directly related to a reduced transient.

The TBL problem is reduced to two separate problems, one to satisfy the throughput criterion and other to reduce periodic-output latency. We provide a solution to the TBL problem by first finding a schedule that satisfies the throughput constraint and then processing the schedule thus obtained using the proposed post-processing technique to obtain the final schedule. The latency-reduction post-processing strategy involves generating schedules by retiming the schedule obtained after phase 1 and choosing the one with minimum periodic-output latency. Retiming techniques have been used to solve a variety of problems such as cycle-time minimization and area minimization [24, 27]. One contribution of this report is to show that retiming also plays an important role in reducing transient and periodic-output latency and to develop efficient retiming techniques that achieve these goals.

If one wants to reduce the transient-interval instead of periodic-output latency after phase 1, one can apply schedule-post processing technique for transient reduction in phase 2. In this report, the approach to reduce the transient is to take feedback from the execution pattern of a schedule obtained from a dataflow graph and refine it. In our proposed technique, the way we extract a schedule from the self-timed execution of an application ensures that in the final schedule the assignment of actors onto the processors would be same, and the order of execution of actors on the processors would be just a shifted version (phase difference) of the order of actors in the schedule at the end of phase 1. Also, the distribution of delays in the final schedule is the same as some retimed version of the schedule obtained by phase 1. One should also note that this schedule post-processing technique is essentially retiming the initial schedule and re-ordering the execution of actors on their respective processors.

## 3. Algorithm and experimental results

In this section, we explain our algorithm for solving the TBL problem or for reducing the transient for a given throughput constraint. This algorithm shows how the schedule post-processing technique for latency-reduction can be used with a known throughput constrained scheduling technique to solve the TBL problem.

The TBL problem has been addressed in two phases. In the first phase, a schedule that satisfies the throughput constraint is found using an approach described in [19]. Any algorithm for throughput-constrained or throughput-minimized scheduling can be used in the first phase. For prototyping purposes, we chose the technique of [19] since it is a particularly well-documented one. In the second phase, we try to minimize periodic-output latency as described in the last section. Specifically, Hoang [19] describes a scheduling technique to schedule a dataflow graph such that the throughput is greater than or equal to a given minimum. This technique has been used as a "subroutine" in our proposed algorithm to solve the TBL and transient-reduction problems. As mentioned above, any other scheduling technique that gives a schedule that satisfies a pre-specified throughput constraint can be used in place of Hoang's technique. Hoang's technique has been used just as an example to explain and experiment with the proposed algorithm that can be used to solve the problem of transient-reduction with a throughput constraint as well as the TBL problem. Thus, our schedule post-processing techniques are quite general in nature and can be applied to schedules to generate modified schedules with smaller transient or periodic-output latency.

## 3.1. Phase 1

To compute a schedule that satisfies the minimum throughput requirement, Hoang's algorithm as described in [19] is used. As we have discussed above, one may use any other suitable technique here for obtaining the initial schedule that satisfies the given throughput constraint.

## 3.2. Phase2

For use in phase 2, we will describe two different schedule post-processing techniques, a transient-reduction technique and a latency-reduction technique. Reducing transient is beneficial for self-timed iterative systems. Settling of the execution pattern fully exposes parallelization in the periodic pattern. Reducing the transient would help one exploit hitherto underutilized parallelism in the application much sooner. To solve the TBL problem, our latency-reduction technique should be applied for phase 2. If the aim is to find a schedule with minimum transient for a given throughput constraint, then one should use our transient-reduction technique for phase 2. In this subsection, first we explain the transient-reduction technique and then the latency-reduction post processing scheme.

### 3.2.1 Transient-reduction scheme

A heuristic based on feedback from simulation is used to reduce the transient in the following way.

The schedule is simulated to execute in a self timed manner and the pattern is observed when it becomes periodic. Since we know that the throughput of the system is greater than or equal to the given throughput constraint, a schedule formed by extracting the order of execution of actors in a given period of the periodic pattern would satisfy the throughput constraint, as a self timed execution can be represented by a Markov process. A Markoff process is a stochastic process whose past has no influence on the future if its present is specified [29]. Since in a self-timed execution also, only the present state influences all the future states in the execution pattern, self-timed execution can be considered a Markoff process. The state at some time instant during self-timed execution is comprised of all the information of the dataflow graph that is pertinent for future execution, such as delay distribution at the edges of the dataflow graph, execution status of every actor of the dataflow graph, etc. So if we start with the schedule derived from the order of execution, at the start of the period of the throughput-constrained schedule, the new schedule also would result in a throughput that is no less than the throughput constraint. The state of the system at the instant when the transient ends, combined with the order of execution of actors during that periodic pattern, is output as the new schedule.

The rationale for this heuristic can be further explained in the following way. A self-timed execution can be represented as a Markoff process. That is, the future execution of the system depends only on the present state of the system. Supposing that time is managed in discrete steps, given the present state, there could be a few states that the system can never reach. This introduces the notion of *distance* between two states in a self-timed execution. One can define distance from a state $S_1$ to another state $S_2$ as the time taken for the system to reach the state $S_2$ from state $S_1$. Note that the distance of $S_2$ from $S_1$ may be different than distance of $S_1$ from $S_2$. So, it could be helpful to know the state of the system when the pattern has become periodic as starting from a state *closer* to this state, that is, starting from a state that is at a lesser distance from this state, could take us to a periodic pattern faster than starting from some arbitrary state. Hence, the state of the system is recorded at the time instant when the pattern has become periodic and a new schedule is formed based on that. Also, minimization of delays in the dataflow graph (e.g., through retiming) may give us smaller transient, as more delays would lead to longer time in generating valid outputs.

One can transform a multiple output node graph into an equivalent single output node graph by introducing an edge between every output node and a new synthetic output node with zero execution time. We assume that an output sample is generated when all output nodes have executed during a particular iteration. So the execution of the output node symbolizes the generation of all output samples for that iteration. A data source, such as a digitized microphone output, that generates samples periodically, is modelled by adding an input actor with edges to all the start actors in the dataflow graph and a self-loop edge with one token and with execution time equal to the input period. A pseudocode for our transient-reduction scheme is shown in Figure 8.

In this algorithm, *calculatePeriod* is a function that takes the execution statistics as input and calculates the time when the transient of the execution pattern ends. *Stats*($S$) represents the execution statistics of schedule $S$. Execution statistics for any schedule can be found by simulating the execution of that schedule and is composed of the token distribution around the edges of the dataflow graph and the information about execution of every actor in the dataflow graph at any time instant. The function *generateSchedule* takes a time instant $t$ and execution statistics as inputs and outputs a schedule in the required format using the state information at time $t$. It finds the order of execution of various actors on processors starting from time $t$ and the distribution of tokens on all inter-processor communication (IPC) edges at that instant. This order of execution of actors on different processors is the order of execution that is followed in the output schedule. In case an actor is executing on some processor at time instant $t$, the order of execution includes that actor as the first one to execute on that processor. The algorithm for *generateSchedule* is described in Figure 9. Note that the final schedule does not alter the assignment of

```
Function transientReduction
Input Schedule $S$
Output Improved schedule $S'$

execute ($S$)
$L$ = calculatePeriod(Stats($S$))
$S'$ = generateSchedule($L$, Stats($S$))
Output $S'$
```

Figure 8. Pseudocode for transient-reduction post-processing.

actors to processors. It only changes the numbers of tokens on IPC edges and the order of execution of these actors on processors.

One way to improve this heuristic is to introduce a few temporary *first-iteration actors* on the processors where some actor $n$ is in the middle of execution at time instant $t$. On those processors for every actor $n$ in the middle of execution at time $t$, we can introduce a first-iteration actor — effectively, a NOP (no-operation) — that executes for the time interval between $t$ and the time when that actor $n$ finishes execution, and does not execute in any of the later iterations. This way, the execution of this new schedule will begin with the state the system was in at the end of the transient. This would reduce the periodic-output latency significantly but the schedule can not be called strictly self-timed as first-iteration actors should be executed only once, whereas all actors in self-timed schedule are executed iteratively.

```
Function generateSchedule
Input Time instant $t$, Stats($S$) of the input schedule $S$
Output Final schedule $S_f$

1. Record the token distribution on all IPC edges at $t$
2. for each processor{
     Record the order of execution of actors starting from
     $t$
     }
3. Output the schedule $S_f$ from the information gathered in
steps 1 and 2.
```

Figure 9. Pseudocode for function *generateSchedule.*

**Example 3:** Figure 10 shows the self-timed execution pattern when first-iteration actors are used for the application graph shown in Figure 2. In the self-timed execution shown in Figure 3, to start with the state at 7 time units (when $E$ finishes executing its first invocation), we introduce first-iteration actors in the schedule as shown in Figure 10. ■

For the results reported in the later part of the report, *generateSchedule*, as described in Figure 9, was used, which does not introduce first-iteration actors.

Figure 11 shows the pseudocode to compute a minimum transient schedule that satisfies a throughput constraint($1/T$). *Phase1Algo* is a function that represents the algorithm used in phase 1. It takes $T$ (inverse of throughput constraint $tr_{min}$) and an application graph as inputs and outputs a schedule.

The multiprocessor architecture that we consider is a bus-based architecture in which a group of processors communicate by means of a single shared bus. Table 1 shows the result for several schedules for four applications: *FFT, QMF, Karp* and *Meas.* These represent, respectively, a fast Fourier transform, a quadra-

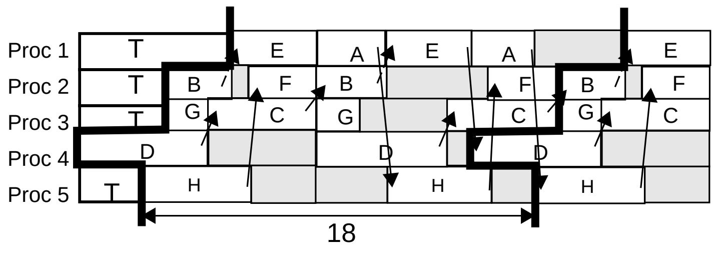


Figure 10. Self-timed execution with first-iteration actors denoted by T.

ture-mirror filter bank, a music synthesis application based on Karplus-Strong technique, and a measurement application. FFT applications are of three types based on examples given in [26]; *fft1*, *fft2* and *fft3*; and each of these value represents an FFT application with a particular set of execution times for actors. In this report, an FFT is application represented as *fftx.y.z,* where *x*, *y* and *z* are integers. The symbol *x* takes values 1, 2 and 3 and each of this value represents one of the FFT applications out of *fft1*, *fft2* and *fft3*. The *y* value represents the inter-processor communication (IPC) cost among actors when the application executes. The *z* value represents how many processors are employed to run the application. Thus, *fft1.1.2* represents an *fft1* application with IPC cost of 1 to be scheduled on 2 processors.

| Application | $T$ | $R_{old}$ | $R_{new}$ | $\mu_r$ |
|---|---|---|---|---|
| fft1.1.2 | 200 | 157 | 150 | .044 |
| fft1.2.5 | 200 | 319 | 298 | .066 |

Table 1. Results of the algorithm described in Figure 11 for various benchmarks with deterministic execution times.

```
Input $T$, Application graph $G$
Output Final schedule $S'$

$S$ = Phase1Algo($G$, $T$)
$S'$ = transientReduction($S$)
Output $S'$
```

Figure 11. Pseudocode to find minimum-transient schedule for a given $T$.

| Application | $T$ | $R_{old}$ | $R_{new}$ | $\mu_r$ |
|---|---|---|---|---|
| fft1.1.5 | 180 | 912 | 739 | .189 |
| fft1.1.10 | 70 | 57 | 56 | .017 |
| fft1.1.10 | 70 | 57 | 78 | -.368 |
| fft1.5.5 | 200 | 364 | 163 | .552 |
| fft1.5.10 | 150 | 1182 | 1064 | .099 |
| fft1.10.10 | 200 | 4463 | 4454 | .002 |
| fft1.10.5 | 150 | 4220 | 3830 | .092 |
| fft2.1.2 | 225 | 1500 | 1400 | .066 |
| fft2.1.10 | 125 | 11641 | 11079 | .048 |
| fft2.5.10 | 150 | 4885 | 4500 | .0788 |
| fft2.10.10 | 210 | 3710 | 2910 | .215 |
| fft2.5.5 | 300 | 1200 | 1070 | .108 |
| fft3.1.2 | 300 | 500 | 500 | 0 |
| fft3.1.10 | 100 | 226 | 225 | .004 |
| fft3.2.2 | 300 | 500 | 400 | .2 |
| fft3.5.5 | 300 | 1000 | 710 | .29 |
| fft3.5.8 | 225 | 1655 | 125 | .924 |
| fft3.10.5 | 300 | 2020 | 1960 | .029 |
| fft3.10.12 | 180 | 6370 | 5770 | .094 |
| | | | | |
| qmf2 | 150 | 400 | 350 | 0.125 |
| qmf3 | 150 | 518 | 483 | .0675 |
| qmf4 | 150 | 612 | 468 | 0.2353 |
| | | | | |

Table 1. Results of the algorithm described in Figure 11 for various benchmarks with deterministic execution times.

| Application | $T$ | $R_{old}$ | $R_{new}$ | $\mu_r$ |
|---|---|---|---|---|
| karp2 | 450 | 800 | 700 | 0.125 |
| karp3 | 450 | 800 | 800 | 0 |
| karp4 | 450 | 900 | 800 | 0.1111 |
| | | | | |
| meas2 | 250 | 700 | 650 | 0.0714 |

Table 1. Results of the algorithm described in Figure 11 for various benchmarks with deterministic execution times.

Similarly, QMF, Karp and Meas applications are represented as *qmfx*, *karpx* and *measx,* where *x* is an integer that represents the number of processors the applications needs to be scheduled on. For example, *qmf2*, *karp3* and *meas2* represent QMF application on 2 processors, a Karp application on 3 processors, and a Meas application on 2 processors, respectively.

The figure of merit $\mu_r$ for the transient-reduction scheme is defined as

$$\mu_r = (R_{old} - R_{new}) / R_{old}, \tag{10}$$

where $R_{old}$ is the time instant when the transient ends in the execution of the schedule input to transient-reduction post-processing. The symbol $R_{new}$ represents the corresponding time for the final schedule, i.e., the schedule output by *transientReduction*. The symbol $T$ denotes the inverse of the minimum throughput requirement, i.e., $T = 1/(tr_{min})$. It can be seen from results tabulated in Table 1 that for systems with deterministic execution times, there is a reduction in the transient in most of the cases. Since our transient-reduction scheme does not use first-iteration actors, as explained earlier, and may inject more initial tokens, it may happen that the schedule generated by the transient-reduction scheme has a worse transient than the original schedule. In one case, the transient increased by 21 time units beyond an already small transient of 57 time units. One can see that out of 28 experiments tabulated in Table 1, there was an improvement of more than 5% in 19 cases, of more than 10% in 11 cases and of more than 20% in 6 cases.

In the case of non-deterministic execution times, the execution does not settle to a periodic pattern unlike systems with deterministic execution times. The notion of a transient has not been defined by us for non-deterministic execution. Though phase 1 of the algorithm can be applied to this case to get a schedule to meet the throughput criterion, it is not meaningful to use phase 2 to reduce the transient in case of non-deterministic execution times because in non-deterministic execution, the execution pattern does not, in general, becomes periodic.

### 3.2.2 Latency-reduction scheme

Unlike the case of the transient, periodic-output latency, as defined in this report, can be calculated both for deterministic and non-deterministic systems. Using the off-line method, which has been described in detail in Section 1.1, given a periodic-output latency $L$ and an output buffer size $B$ for a system, the throughput of the system can be calculated and vice-versa. The results of scheduling with an aim to reduce periodic-output latency, using a technique that is conceptually very similar to the transient-reduction technique, are tabulated in Table 2. Here, after getting an initial schedule after phase 1, the average iteration period of the execution pattern is chosen as $T$ and $L$ is found for that $T$ in phase 2. The state of the system at the end of $L$ is used to find out the final schedule in the same way as is done in transient-reduction scheme. We introduce a few definitions that are useful in explaining the experimental results. In case of non-deterministic execution times in an application, execution times for various actors may vary from one execution iteration to the other because of data-dependency of execution times, cache misses, interrupts, etc. The number of possible execution times taken by an actor $i$ is denoted by $n_i$. We denote the set of $n_i$ possible execution times taken by an actor $i$ as $\{t_{i1}, t_{i2}, \ldots, t_{in_i}\}$. The probability of occurrence of a possible execution time $t_{ik}$ for actor $i$, is denoted by $p_{ik}$, for all $k = 1, \ldots, n_i$. The *degree of non-determinacy* $\lambda$ is a measure of overall amount of non-determinacy in the application, specifically, in the actor execution times, and is defined as

$$\lambda = \frac{\sum_i \left\{ \sum_{k=1}^{n_i} \{ p_{ik}(t_{ik} - t_{i,mean})^2 \} \right\}}{\sum_i \{ t_{i,mean} \}^2}, \tag{11}$$

where $t_{i,mean}$ denotes the mean execution time of actor $i$ and is defined as

$$t_{i,mean} = \left( \sum_{k=1}^{n_i} t_{ik} \right) / n_i . \tag{12}$$

The figure of merit $\mu_l$ for latency-reduction is defined as

$$\mu_l = (L_{old} - L_{new}) / L_{old}, \tag{13}$$

where $L_{old}$ and $L_{new}$ are the periodic-output latencies of the schedule input to latency-reduction post-processing and the final output schedule respectively, for a given $T$ and $B$.

One can see from Table 2, that in case of non-deterministic execution times, in about half of the cases, this approach does not lead to a low-latency schedule. This is expected, as in our approach we are trying to reduce latency by starting from a state that is expected to be closer to the state at $L_{old}$, but, because the execution times are non-deterministic, this approach of starting from some better state fails. We present a new latency-reduction post processing strategy that can be applied in phase 2 to solve TBL problem. The pseudocode for this improved latency-reduction post-processing approach is shown in Figure 12.

Here, we try to find the best possible retiming of the schedule obtained after

| Application | $\lambda$ | T | B | $L_{old}$ | $L_{new}$ | $\mu_l$ |
|---|---|---|---|---|---|---|
| fft1.5.2 | .058 | 201 | 5 | 325 | 419 | -.289 |
|  | .146 | 212 | 5 | 277 | 308 | -.111 |
|  | .264 | 212 | 5 | 549 | 846 | -.541 |
| fft1.5.5 | .058 | 179 | 5 | 375 | 648 | -0.728 |
|  | .146 | 178 | 5 | 997 | 821 | 0.176 |
|  | .264 | 174 | 5 | 3147 | 3147 | 0 |
| fft1.5.10 | .058 | 106 | 5 | 182 | 288 | -0.582 |
|  | .146 | 105 | 5 | 425 | 385 | 0.094 |
|  | .264 | 103 | 5 | 337 | 604 | -0.792 |

Table 2. Results for latency-reduction using a method similar to transient-reduction approach for FFT graphs with non-deterministic execution times.

phase 1, from the point of view of reducing latency. We do this by choosing the actor whose retiming reduces the latency most, which is found by simulating the retimed schedule. By doing this repetitively, until there is no improvement, we end up with a schedule that is a retimed version of the original schedule, but with lower latency.

With respect to the pseudocode given in Figure 12, a *positive retiming step* [12] for an actor is defined as “removing one delay from each of the edges to successor actors and adding one delay to each edge from the predecessor actors.” Similarly, a *negative retiming step* is “removing a delay from each of the incoming edges to the actor and adding one delay to each of the outgoing edges from the actor.” A positive or negative retiming step is called a *legal retiming step* if no edge has a negative delay on it after retiming.

**Example 4:** Figure 13 illustrates these concepts. Figure 13(a) shows an example dataflow graph. The values alongside the edges represent the numbers of tokens on them. Figure 13(b) shows the distribution of delays on the edges after a legal positive retiming step on actor D of the dataflow graph. The result of a legal negative retiming step on actor D of the dataflow graph is shown in Figure 13(c). A negative retiming step on actor B of the data flow graph is shown to be illegal in Figure 13(d) as it leaves a negative delay on one of the dataflow graph edges. ■

Using the off-line latency-computation method, which has been described in detail in Section 1.1, given a throughput and output buffer size $B$ for the system, the periodic-output latency of the system can be calculated and vice-versa. Function *findL* is a function that outputs the periodic-output latency for schedule $S$ for a given

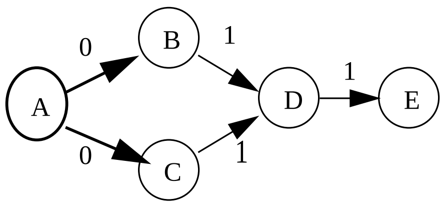
$T$ and $B$ using the off-line latency-computation method.

Pseudocode that gives our approach to solving the TBL problem is shown in Figure 14. Experiments were conducted to test the efficacy of this approach in solving the TBL problem. The results are tabulated in Table 3. From the results for

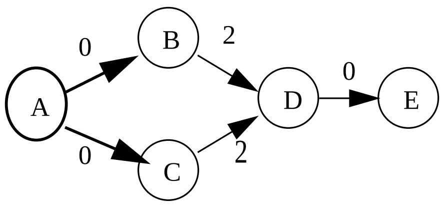
```
Function latencyReduction
Input Schedule $S$, $T$, $B$
Output Final schedule $S_f$

presentL=findL($S$, $T$, $B$)
$L_{min}$ = 0;
while($L_{min}$ < presentL){
      $R$= set of schedules obtained after applying any
      legal retiming step on $S$.
      for each $S_i$ belonging to $R$ {
            if $S_i$ satisfies throughput constraint {
                  $L_i$ = findL($S_i$, $T$, $B$);
            }
      }
      $L_{min}$ = minimum of all $L_i s$;
      $S_{min}$ is the schedule corresponding to $L_i$.
      if ($L_{min}$ < presentL){
            $S$ = $S_i$;
            presentL = $L_{min}$;
      }
}
$S_f$ = $S$;
Output presentL and $S_f$
```

Figure 12. Pseudocode for latency-reduction post-processing.

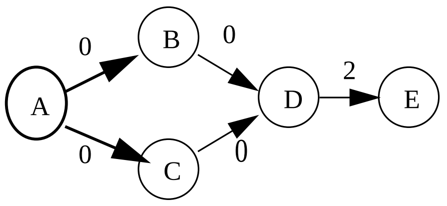
benchmarks *fft1.5.5*, *qmf4* and *karp10* and *meas2* in Table 3, one can see that the heuristic works well and there is a significant improvement in periodic-output



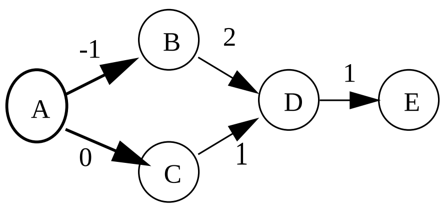
(a) A dataflow graph



(b) A legal positive retiming step on actor D



(c) A legal negative retiming step on actor D



(d) An illegal retiming step on actor B

Figure 13. Illustration for retiming.

latency.

| Application | $\lambda$ | $T$ | $L_{old}$ | $L_{new}$ | $\mu_l$ |
|---|---|---|---|---|---|
| fft1 | 0 | 180 | 185 | 185 | 0 |
| fft1 | .0586 | 181 | 382 | 189 | .5052 |
| fft1 | .1171 | 179 | 614 | 180 | .7068 |
| fft1 | .2347 | 173 | 885 | 741 | .1627 |
| fft1 | .3656 | 158 | 771 | 647 | .1608 |
| fft1 | .3594 | 176 | 971 | 971 | 0 |
| fft1 | .5245 | 191 | 1022 | 797 | .2201 |
| fft1 | .6059 | 167 | 3197 | 904 | .7172 |
| | | | | | |
| qmf | 0 | 124 | 125 | 125 | 0 |
| qmf | .0123 | 123 | 685 | 419 | .3883 |
| qmf | .0844 | 123 | 575 | 575 | 0 |
| qmf | .1453 | 125 | 778 | 778 | 0 |
| qmf | .2561 | 113 | 4112 | 3099 | .2463 |
| qmf | .318 | 129 | 2260 | 1026 | .5456 |
| qmf | .3928 | 122 | 3699 | 1777 | .5201 |

Table 3. Results of applying *latencyReduction* to non-deterministic graphs.

```
Input $T$, Application graph $G$
Output Final schedule $S'$

$S$ = Phase1Algo($G$, $T$)
$S'$=latencyReduction($S$, $T$, $B$)
Output $S'$
```

Figure 14. Pseudocode to solve the TBL problem.

| Application | $\lambda$ | $T$ | $L_{old}$ | $L_{new}$ | $\mu_l$ |
|---|---|---|---|---|---|
| qmf | .4748 | 123 | 2962 | 2082 | .2971 |
| qmf | .5684 | 121 | 4779 | 4779 | 0 |
| qmf | .72 | 120 | 5226 | 3773 | .2780 |
| | | | | | |
| karp | 0 | 400 | 643 | 643 | 0 |
| karp | .0380 | 400 | 721 | 605 | .1609 |
| karp | .1364 | 386 | 2411 | 1231 | .4894 |
| karp | .2139 | 394 | 1826 | 1289 | .2941 |
| karp | .3096 | 394 | 1663 | 1329 | .2008 |
| karp | .4572 | 400 | 4398 | 899 | .7956 |
| karp | .5356 | 400 | 3140 | 565 | .8200 |
| karp | .6080 | 371 | 3302 | 1803 | .4539 |
| | | | | | |
| meas | 0 | 182 | 217 | 217 | 0 |
| meas | .0537 | 192 | 1353 | 773 | .4286 |
| meas | .1022 | 181 | 3241 | 2294 | .2922 |
| meas | .2208 | 174 | 5964 | 5676 | .0483 |
| meas | .3111 | 189 | 5572 | 5343 | .0411 |
| meas | .4240 | 181 | 7789 | 7700 | .0114 |
| meas | .5681 | 173 | 9531 | 9510 | .0022 |
| meas | .6066 | 171 | 10348 | 10348 | 0 |

Table 3. Results of applying *latencyReduction* to non-deterministic graphs.

Figure 15 shows plots of $\mu_l$ vs. $\lambda$ corresponding to Table 3 for these benchmarks. It can be seen that the improvement does not follow a monotonic trend, which is understandable as the algorithm finds out the best retiming after taking feedback from simulation and is not affected by the degree of non-determinacy in some specific way.

To study the relationship between buffer size ($B$), throughput $(1/T)$ and periodic-output latency ($L$) using the off-line method for latency-computation, we generated a schedule for *fft2.5.5* with $\lambda = 0.326$ by using *Phase1Algo* and a throughput constraint of $1/265$. The value of $T$ in the off-line computation was then varied around 265 for several values of $B$, and $L$ was computed using the off-line method. Figure 16 shows the relationship between $L$ and $T$. It can be seen that $L$ is minimum for a $T$ value close to 265 for almost all $B$ values, and as the deviation of $T$ from 265 increases, $L$ also starts increasing. One can also observe that for an increase in $B$, $L$ deceases for the same value of $T$, as explained in Section 1.1.

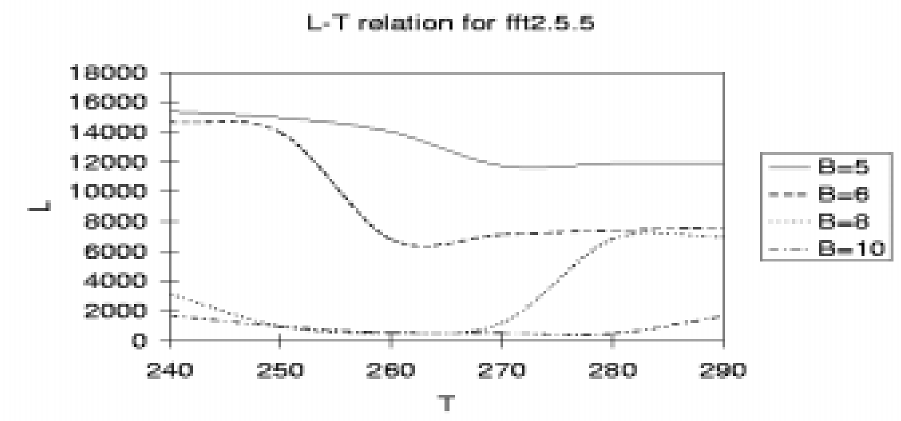


Figure 16. $L - T$ plot for *fft2.5.5* with $\lambda = 0.326$.

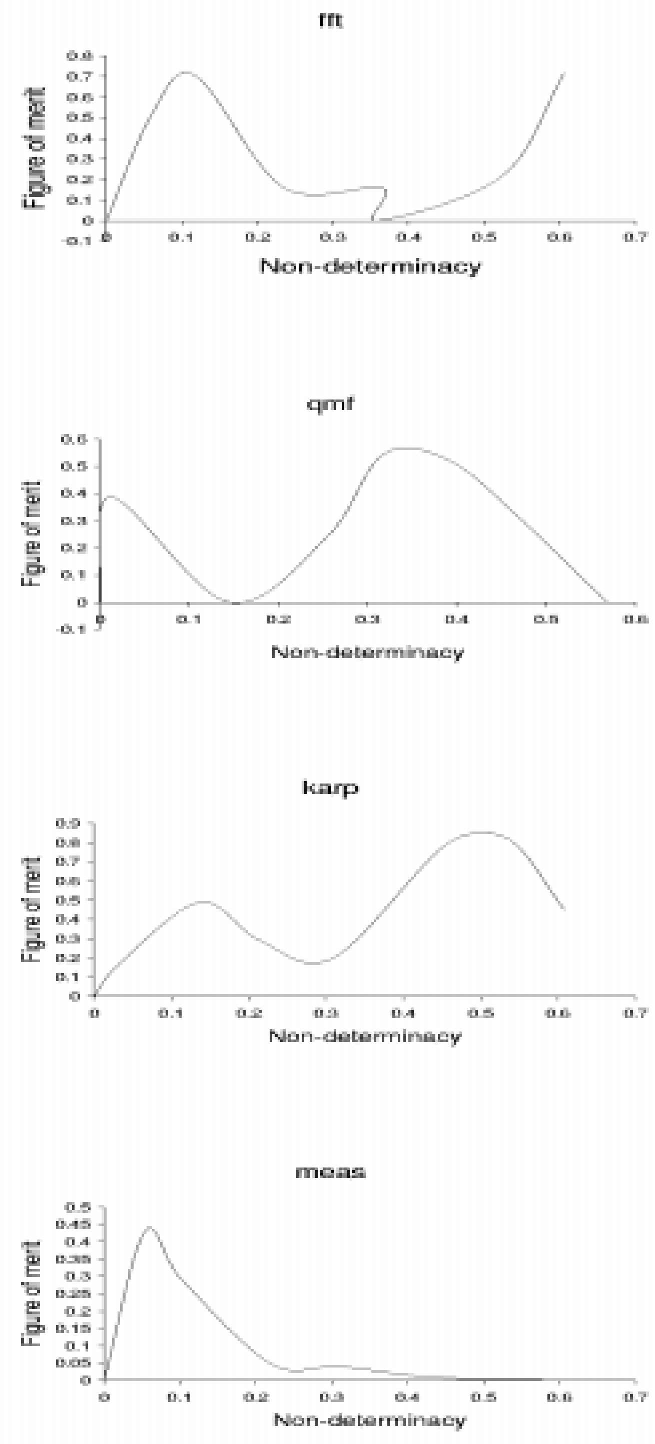


Figure 15. Plots of $\mu_l$ vs. $\lambda$ for FFT, QMF, Karp and Meas corresponding to Table 3.

## 3.3. Summary

Schedule post-processing strategies for transient-reduction and latency-reduction are described in this section. An algorithm to solve TBL problem that uses one of the schedule post-processing strategies is also presented. Experimental results are presented that show the efficacy of the proposed strategies.

# 4. Streamlining latency analysis for contention-free systems

In this section, we present a new approach to implement the proposed schedule post-processing techniques using a graph-theoretic framework, unlike the usual approach of event-driven simulation. In practical multiprocessor systems, there is contention for shared communication resources. One example of this is a shared bus. A processor must gain access to the shared bus before it can execute an IPC operation. The analysis we present here applies only to contention-free multiprocessors such as a fully-connected multiprocessor system. In mesh architectures, such as the Raw architecture [40], processors are connected directly to their immediate neighbors in a mesh-design and hence the system does not in general have as much contention as a shared-bus system. So mesh architecture-based multiprocessors often result in a contention-free systems. We first introduce a few relevant definitions and then present recurrence relations that can be used to find starting times of actors for different invocations during execution and hence simulation using an event-driven simulator is not needed. The following analysis applies only to applications with deterministic execution times on contention-free systems. We define a *deterministic, contention-free self-timed system (DCFS)* as a system where there is no contention for communication resources, the application actors have deterministic execution times, and the execution of the application is carried out in a self-timed manner. We model the application in a DCFS as an IPC graph, which is described in

Section 1.1. We model periodic excitation (input) by modeling the source node $v_s$ as having an execution time equal to the sample period $T_s$, and adding a self-loop edge $(v_s, v_s)$ with unit delay to the IPC graph $G_{ipc}$ for the application. Also the sample period $T_s$ is equal to the maximum cycle mean of the graph (a reasonable assumption because otherwise, there will be data loss or unbounded accumulation on the system input buffers). In a DCFS, the communication actors in the IPC graph of the application can execute simultaneously as there is no contention for communication resources in a DCFS.

This approach reduces the execution time of the presented schedule post-processing techniques considerably for a DCFS.

## 4.1. Notation and analysis

Let the execution time of an actor $v$ be denoted by $\tau(v)$. We define a path as an alternating sequence of edges and nodes in a directed graph, such that the node following an edge in the sequence is its sink vertex, and the sequence starts on an edge and ends on a node. The last node in the sequence is called an *end node* of the path. For example, for an integer $n$, $(e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n)$ represents a path, where $e_k, \quad k = 1, \ldots, n$, denote a set of edges in the directed graph, and $v_k, \quad k = 1, \ldots, n$ denote a set of vertices in the directed graph such that $v_k = \text{sink}(e_k), k = 1, \ldots, n$. Node $v_n$ is the end node of the path in this example.

Suppose a path $p$ consists of a sequence of edges and nodes represented by $(e_1, v_1, e_2, v_2, \ldots, v_{n-1}, e_n, v_n)$. Define $\eta(p) = \text{delay}(e_1)$. If $\eta = 0$, then the path has 0 delay. For $\eta > 0$, path $p$ is a $k$-*delay path* for some integer $k > 0$ if it satisfies the following two conditions.

$$1)\text{ For } \eta = 1, \quad \sum_{i=1}^{n} \text{delay}(e_i) = k \quad ; \tag{14}$$

$$2)\text{ For } \eta > 1, \quad \sum_{i=1}^{n} \{\text{delay}(e_i)\} \geq k \geq \sum_{i=1}^{n} \{\text{delay}(e_i)\} - (\eta - 1) \text{ .} \tag{15}$$

Furthermore, if a path $p$ is a $k$-delay path, then delay of the path can be called equal to $k$, i.e. $\text{delay}(p) \equiv k$. This implies that the delay of path $p$ may be multi-valued if $\eta > 1$. This is to account for the fact that if $\eta(p) > 1$, then the delay of path $p$ can be chosen as any number from $\sum_{i=1}^{n} \text{delay}(e_i)$ to $\sum_{i=1}^{n} \{\text{delay}(e_i)\} - (\eta - 1)$ as now one can choose number of tokens in the range of $1$ to $\eta$ on $e_1$, to be included in path $p$. Hence the delay of path $p$ may be multiple-valued.

The execution time of the path $p$, denoted by $t_{path}(p)$, is defined as the sum of the execution times of the nodes on the path minus the execution time of the end node. That is, $t_{path}(p) = \tau(v_1) + \tau(v_2) + \ldots + \tau(v_{n-1})$.

Let $\theta(v, k)$ denote the length of a longest (maximum execution time) $k$-delay path directed to $v$. More precisely,

$$\theta(v, k) = max(\{t_{path}(p) | ((p \text{ is a directed path to } v) \text{ and } (\text{delay}(p) = k)) \}) \text{.} \tag{16}$$

Note that from the definition of the $max$ operator, $\theta(v, k) = 0$ if there is no $k$-delay path directed to $v$.

Suppose that in a DCFS, the starting time of the $k$ th invocation of an actor $a$ is denoted by $t_k(a)$.

**Lemma 1:** The starting time of the $k$ th invocation [32], $t_k(a)$ of an actor $a$ in a DCFS, is equal to the length of the longest $k$-delay path directed to $a$, assuming actor invocations are numbered starting at 1. That is,

$$t_k(v) = \theta(v, k) \text{ for } k = 1, 2, 3, \ldots \tag{17}$$

**Proof:** This results follows naturally from developments in [32]. The evolution of the network can be described by the following equations ((19), (20), (21)). Here $IN(s)$ represents the set containing all the immediate predecessors of actor $s$ and $(a, b)$ denotes the edge directed from an actor $a$ to an actor $b$. (18)

$$t_k(v) = 0 \text{ for } (k < 1) \text{, for all actors } v; \tag{19}$$

$t_1(s) = 0$, where $s$ is an actor that satisfies either of the following two conditions.(20)

1. $IN(s) = \phi$, or
2. for all $u \varepsilon IN(s)$, $\text{delay}(u, s) \geq 1$;

$$\text{and } t_k(v) = max_{u \varepsilon IN(v)}\{l_k(u, v)\} \text{ for all actors } v, \tag{21}$$

$$\text{where } l_k(u, v) = \begin{cases} t_{k-\delta}(u) + \tau(u) \text{ if } (k > \delta) \\ 0 \text{ otherwise} \end{cases} \tag{22}$$

and $\delta = \text{delay}(u, v)$.

It is easy to see that for every actor $v$, $t_1(v) = \theta(v, 1)$.

Assume for $k \geq 1$, (17) holds for an actor $v$.

Now divide the set $IN(v)$ into sets $IN_1(v)$ and $IN_2(v)$ such that for all $u \varepsilon IN_1(v)$, $k + 1 > \mathrm{delay}(u, v)$ and for all $u \varepsilon IN_2(v)$, $k + 1 \leq \mathrm{delay}(u, v)$. Now, using (21),

$t_{k+1}(v) = max_{u\varepsilon\{IN_1(v) + IN_2(v)\}}\{l_k(u, v)\}$. Since $max_{u\varepsilon IN_2(v)}\{l_k(u, v)\} = 0$, we have that

$$t_{k+1}(v) = max_{u\varepsilon IN_1(v)}\{t_{k+1-\delta}(u) + \tau(u)\}$$

$$\Rightarrow \quad t_{k+1}(v) = max_{u\varepsilon IN_1(v)}\{\theta(u, k + 1 - \delta) + \tau(u)\}$$

$$\Rightarrow \quad t_{k+1}(v) = \theta(v, k + 1)$$

So, by induction, (21) is true for all $k \geq 1$. Q.E.D.

## 4.2. Results

The approach to use the recurrence relations (19), (20), (21) to find the starting times of all invocation for the output actor gives us a much faster way to implement the “simulation” needed in the proposed schedule post-processing techniques. Table 4 tabulates the speedups observed by “simulation” using this approach compared to usual even-driven simulation approach. One can see that significant speedups are achieved for various DSP benchmarks, when (19), (20), (21) are to calculate the generation of output samples during execution instead of using event-driven simulation.

# 5. Problem formulation and overview of model

In polymorphous computing architectures (PCA), various attributes of the architecture can be varied, such as inter-processor message routing, caching policies, scheduling policies, processor voltages, resource allocation to computing units, and architectural support for synchronization during inter-processor communication. A polymorphous computing architecture can be a particularly useful platform for developing a computing system where applications and the performance requirements keep changing as one can adaptively configure the PCA to suit the dynamic constraints and objectives. In this section, we first define a problem that deals with execution of an application on a polymorphous computing architecture such that the specified performance requirements are satisfied, where performance requirements

| Application | Speedup |
|---|---|
| fft1 | 34 |
| fft2 | 48 |
| fft3 | 52 |
| qmf | 25 |
| karp | 42 |

Table 4. Speedups for various benchmarks.

may vary over time and the application may have tasks with stochastic execution times. Then we present a general model as a solution to the problem and this model is shown to be efficient and powerful enough to be able to handle diverse applications, through analysis and experiments.

The actors in the application are assumed to have stochastic execution times with distributions that may vary slowly over time. The computing unit is a reconfigurable architecture, and we have to find a mapping of the actors in the application onto the processors in the reconfigurable architecture and the configuration that the architecture should assume, such that all performance-related constraints (e.g., constraints on power, resource usage or throughput) are satisfied and objectives (e.g., maximizing throughput or minimizing latency) are optimized. Henceforth, we will refer to this problem as the *polymorphous computing architecture mapping (PCA mapping) problem.* Moreover, performance requirements (i.e., sets of objectives and constraints), can also change during the execution. As can be seen, the PCA mapping problem is quite general in nature and even very restricted special cases can be proved to be NP-complete.

The approach suggested in this report, is also very general in nature and can handle diverse applications and different performance requirements. It is based upon taking feedback from the execution of the application and/or using mappings computed during earlier executions and modifying the mappings adaptively. It is to be noted that one can not apply relatively sophisticated mapping strategies during the execution of the application as those techniques will take away excessive computational resources away from the application itself. To address this trade-off (thoroughness of dynamic optimization vs. resources drained from the application), our model of the PCA mapping problem also accounts for the time spent in computing efficient adaptations of mappings at run-time on the basis of feedback obtained from execution and identification of bottlenecks, and hence always tries to move towards optimal solution. All the reported experiments were done on an abstraction of the Raw architecture [40].

The Raw microprocessor is a set of interconnected tiles, each of which contains instruction and data memories, an arithmetic logic unit, registers, configurable logic, and a programmable switch that supports both dynamic and compiler-orchestrated static routing [40]. The tiles are connected with programmable, tightly integrated interconnects. Each tile supports multigranular (bit-, byte- and word-level) operations and programmers can use the configurable logic in each tile to construct operations uniquely suited to a particular application. This high degree of configurabilty in the Raw processor system makes it a good choice for PCA in our problem.

In this report, to experiment on our model for the PCA mapping problem, we used an abstraction of the Raw architecture that incorporates salient features of the Raw architecture such as the programmability of interconnects between processors. For experiments, the self-timed execution of applications on this abstracted Raw architecture was simulated. This self-timed execution on the Raw architecture was simulated using the IPC graph model as described in Section 1.1.

## 5.1. Problem formulation

A set of relevant metrics, such as {latency, throughput, average power, peak power, number of resources,...}, is denoted by $M$. If a certain metric appears as a constraint with a *constraint value* to be satisfied when the application executes, then this metric is referred to as a *constraint metric* and the value as a constraint value for that particular metric. A constraint value belongs to the set of real numbers. A pair of constraint metric and constraint value is called a *constraint pair.* For example, $(m, c)$ denotes a constraint pair, where $m$ is a constraint metric and $c$ is the corresponding constraint value. A sequence of constraint pairs is referred to as a *constraint vector,* and is denoted by $V = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K)]$, where $m_1, m_2, \ldots, m_K$ represent any $K$ metrics in $M$, and $c_1, c_2, \ldots, c_K$ represent the corresponding constraint values, for $K \varepsilon \{0, 1, \ldots, N\}$, where $N$ is the number of all constraint pairs. This sequence of constraint pairs in a constraint vector is prioritized such that a $(m_i, c_i)$ is a higher priority constraint pair than a constraint pair $(m_j, c_j)$ if $i < j$, for $i, j \in \{0, 1, \ldots, N\}$ in a constraint vector

$$V = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K)]\,. \tag{23}$$

That is, a constraint pair that appears earlier in the sequence of constraint pairs in a constraint vector has higher priority than the one that appears later. Note that according to this definition, the same metric may appear in many constraint pairs within a given constraint vector. A metric $m_R$ that is to be optimized after all constraints have been satisfied is called a *residual objective*. A *goal* $g$ is an ordered pair

$(V, m_R)$ where $V$ is the constraint vector and $m_R$ is a residual objective. If there is no residual objective, then the goal is composed of only a constraint vector and can be represented by $(V, \perp)$. Here $\perp$ represents the absence of a residual objective. Also, without loss of generality, the metrics are such that the associated optimization problems are to *minimize* the metric (i.e., a lower value of a metric is always better than a higher value). Thus for any metric, if $x, y$ are values of the metric, then $x \le y$ denotes that value $x$ is no worse than value $y$ for the metric. Metrics for which a higher value is more desirable can be transformed into some other metric for which lower value is the better one. For example, the throughput (average rate of completion of application iterations) can be re-cast as the *iteration period*, which is the reciprocal of the throughput.

**Example 5:** Consider a set of relevant metrics $M = \{L, P, T\}$, where $L$ is the latency, $P$ is the average power consumption, and $T$ is the iteration period. Consider the goal $g = [(L, 50), (P, 100), (L, 40), (P, 70), T]$. In $g$, the constraint pair $(L, 50)$ has higher priority than the constraint pair $(P, 100)$, which in turn has higher priority than the constraint pair $(L, 40)$. The metric $T$ is the residual objective. ■

Mapping an application to a reconfigurable architecture includes defining a task-to-processor mapping along with defining the configuration of the reconfigurable architecture. In this report, the scope of the word “configuration” is expanded to include also the mapping of the application onto the reconfigurable architecture. Therefore, a *configuration* consists of two components 1) task-to-processor mapping and 2) configuration of the architecture. Henceforth, the word “configuration” is used in the above sense, unless stated otherwise. A given application, goal, and resource set define an *instance* of the PCA mapping problem. Input to the model is an instance that may change with time. We define the *design space* as the set of all feasible combinations of an instance and a configuration. The *solution space* for a feasible instance is the set of all feasible configurations for that instance. Latency, throughput, average power and peak power are some of the commonly encountered metrics. With many metrics of simultaneous relevance, the goal space is too vast to be fully explored before run-time. The overall high-level view of a model is illustrated in Figure 17. The main components of the model are the *off-line refinement part,* the *configuration store,* and the *on-line refinement part*.

For a given instance, not every configuration is suitable as some configurations may violate constraints or may not fully achieve the optimal objectives. As the goal changes for a given application, one needs to have a suitable configuration specifying the task-to-processor mapping and the architecture. This problem could be undecidable, in general. Also, reconfigurability of the architecture and the stochastic variance of execution times make the solution space consisting of all possible configurations for the input of a goal and a given application much more large. Since computing a suitable configuration is performed during the execution of the applications, it adds to run-time. Hence, exhaustive search strategies are ruled out during run-time. In contrast, one needs low-complexity algorithms for finding these configurations as the goal changes. This is taken care of by the on-line refinement part of the model. It consists of algorithms that find configurations for a given instance. It also consists of feedback units shown by *Identify bottlenecks* block in Figure 17, that takes feedback from the execution of the configurations and modify the configurations so as to better suit the goal, at regular intervals of time.

A configuration store is used to store points in design space that have been explored, so that one can use them later as need be. In [9], Budenske et al. use a similar concept to store relevant data. The off-line refinement part of Figure 17 consists of high-complexity algorithms that yield better solutions. It is acceptable for them to

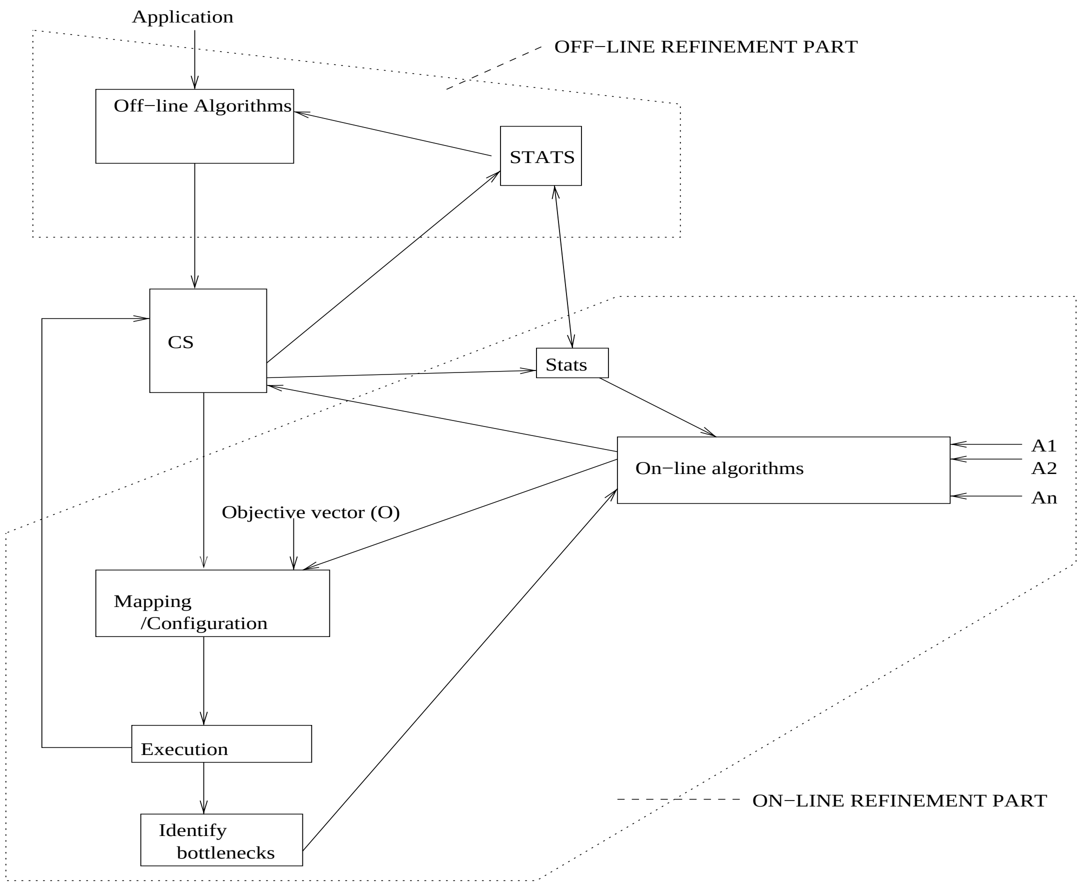


Figure 17. Overview.

be of high-complexity as they are used off-line and do not compete for resources with the application. In Figure 17, the *STATS* unit stores statistics about application (e.g., distributions of execution times for different actors, frequencies of occurrence of some particular regions of goal space, etc.). Off-line algorithms use these statistics for exploring the solution space for input instances.

As soon as the goal or application changes, a search for a suitable configuration in the configuration store is performed. In case this search is “unsuccessful” (all constraints in the goal vector are not satisfied or the residual objective is not fully optimized), on-line algorithms generate a configuration to start with and dynamically refine this initial objective. If a suitable configuration exists in the configuration store, then it is used as a configuration to start with. On-line algorithms keep improving the configuration that is being executed, using feedback from execution. In the meantime, off-line algorithms keep exploring areas of design space and merge the relevant information into the configuration store. They are high-complexity algorithms that use histories of profiling information and generate better configurations for an instance and store them in configuration store, so that they can be picked up directly from configuration store in case the same instance, or a similar one is applied later. The *Stats* unit in the on-line refinement part of the model stores short-term statistics that can be used by on-line algorithms.

## 5.2. Model

The overview of the model shows that it is very adaptive in nature and hence is suitable for applications with stochastic execution times and time-varying goals.

For a practical and robust implementation of the model, a detailed formulation of the model is needed. There are still many issues to the model, such as finding out a good measure to evaluate configurations for a given instance, choosing the instances whose configurations should be stored in configuration store, good low-complexity algorithms that can be used at run-time to find configurations, etc.

These issues are discussed in the next section in a general sense and related problems are formulated in terms of some of well-studied mathematical problems.

# 6. Details of the model

This section deals with the detailed description and algorithms for management of various components of our model for the PCA mapping problem. We have developed a configuration management framework that is the core to the online refinement part of the model of Figure 17, and demonstrated this framework using simple, low-complexity online algorithms for optimization of various metrics. We have showed the efficacy of our model by experimentation on several benchmarks for various goals. We have also analyzed the complexity of various problems related to configuration management by modeling them as some well-studied problems.

## 6.1. Evaluation measure

We need to define some measure of how well a given configuration executes for a particular instance. This evaluation measure should allow unambiguous comparison between the qualities of two configurations based on the current goal.

Suppose we are given a goal $g = [V, mR]$, where

$$V = [(m_1, c_1), (m_2, c_2), \ldots, (m_n, c_n)] . \tag{24}$$

We define the *quality* of a system configuration $C$ as the ordered pair $Q(C) = (k, v)$, where $k + 1$ is the index of first unsatisfied constraint in the constraint vector of that instance, and $v$ is the value obtained for the metric $m_{k+1}$. If configuration $C$ satisfies all constraints in the constraint vector then

$Q(C) = (n+1, v_R)$, where $v_R$ is the value obtained for the residual objective $m_R$ if $m_R \neq \perp$ or $v_R = -\infty$ if $m_R = \perp$. In the following discussion, we assume that the individual metrics over which goals are defined are totally ordered relations (Definition 4 in Section 6.2).

In summary, the quality of a configuration is a measure of evaluation of the configuration with respect to a given instance, and given an instance and two configurations $C_1$ and $C_2$ with qualities $Q(C_1) = (k_1, v_1)$ and $Q(C_2) = (k_2, v_2)$ for that instance, respectively, $C_1$ has higher quality than $C_2$ if

$$(k_1 > k_2) \text{ or } ((k_1 = k_2) \text{ and } (v_1 < v_2)). \tag{25}$$

$Q(C_1) > Q(C_2)\big|_I$ represents that configuration $C_1$ is of higher quality than configuration $C_2$ for instance $I$.

**Example 6:** Consider an application with the relevant metrics being latency ($L$), average power ($P$), and the average iteration period of the output samples ($T$). Let goal $g_1$ = [($L$, 50), ($P$, 90), $T$], and goal $g_2$ = [($L$, 70), ($P$, 100), $T$]. Let $I_1$ denote an instance of the problem, which is composed of the given application, the resource set associated with the PCA, and the goal $g_1$. Similarly, let $I_2$ denote an instance of the problem that is composed of the given application, the resource set associated with the PCA, and the goal $g_2$. Let $C_1$ and $C_2$ be two configurations, each of which denotes a combination of a specific configuration of the architecture and a specific schedule according to which application should be executed on that configuration of the architecture. Suppose further that the attributes of $C_1$ are $L$ = 60, $P$ = 100, and

$T = 40$; and the attributes of $C_2$ are $L = 50$, $P = 100$, and $T = 45$. Qualities of these configurations for instances $I_1$ and $I_2$ are as follows.

$Q(C_1)|_{I_1} = (1, 60)$ and $Q(C_2)|_{I_1} = (2, 100)$

$Q(C_1)|_{I_2} = (3, 40)$ and $Q(C_1)|_{I_2} = (3, 45)$

Therefore, $Q(C_2) > Q(C_1)|_{I_1}$ and $Q(C_2) < Q(C_1)|_{I_2}$. ■

## 6.2. Configuration store

A configuration store serves as a repository of configurations for distinct instances. The allocation of memory in a configuration store to configurations of instances is a fundamental problem in the management of configuration stores. In this section, we develop models to solve this problem.

A configuration store can be divided into several sub-stores (sub-CSs), one for each application. One can define a weight for each application depending upon how often that application is executed and the size of the configurations corresponding to that application. The size of the sub-CS for an application can be directly proportional to the weight of that application. Each sub-CS has some configurations stored in it, one for a specific combination of goal and resource set. In the later part of this section, we assume that we are dealing with a fixed application and a fixed resource set, unless stated otherwise. This does not detract from the generality of the ideas developed later as they can be generalized to include various applications and resource sets using the hierarchical model of configuration store explained above. Using this hierarchical model, a sub-CS would store configurations for various goals for a particular application and resource set. Assuming a fixed resource set, the problem of finding a minimum size configuration store and the goals whose configurations should be stored can be decomposed into several problems, one for each application, of finding a minimum size sub-CS for each application. The minimum size of a configuration store can be found by adding all the minimum size sub-CSs thus found. So, the solution to the problem of finding a minimum size configuration store is based on the solution to the following problem.

*For a particular application and resource set, find the minimum size of sub-CS required for that application and the configurations to be stored in it.*

To address this problem, we consider issues related to storing configurations in a configuration store corresponding to different goals for a fixed application and resource set.

It is beneficial to have a configuration stored in a configuration store appropriate to the present instance as in this case it can be picked up directly from the configuration store. If no appropriate configuration is found in the configuration store then the most suitable of the ones that are present can be picked and on-line algorithms can be employed to compute a more appropriate configuration for the applied instance. Assuming a fixed application and resource set, selecting the goals whose corresponding configurations should be stored in the configuration store depends on various factors such as the size of the configuration store; optimality of stored configuration; computational resources drained from the application; the expected or observed frequency of specific goals, etc. To make the analysis more precise, we first define a few terms and formulate problems related to storage of configurations in a configuration store. These problems relate to various aspects of configuration management such as size; choice of configurations to store; and choice of initial configuration in the configuration store for a given goal. These problems and our models to solve them provide fundamental analysis of the complexity of configuration management and provide feasible, low-complexity solutions to this problem.

### 6.2.1 Terminology and notation

**Definition 1:** Given two goals $g_1$ and $g_2$, we say that $g_1$ is *acceptable* for $g_2$, denoted $g_1 \rightarrow g_2$, if a configuration that satisfies $g_1$ is an acceptable implementation for $g_2$. If $g_1 \rightarrow g_2$, we can also say that $g_1$ *covers* $g_2$. Given a set $\Gamma$ of goals and a specific goal $g$, the *space* of $g$ over $\Gamma$ (or simply, the *space* of $g$, if $\Gamma$ is understood) is $\{g' \in \Gamma | g \rightarrow g'\}$. Thus, the space of a goal $g$ is the set of goals that are acceptably implemented by any configuration that satisfies $g$. The space of a goal $g$ is represented by $space(g)$.

This notion of *acceptability* and *cover* emerge very naturally from the PCA mapping problem and guide the construction and adaptation of the configuration store in our model. In this section, we will also explain how these concepts help us model configuration management and how various results from the analysis of these concepts are useful to the configuration management process.

If one observes that goals from a particular region of the goal space occur more often, one may want to have more configurations stored that satisfy goals in that region of goal space. For some metrics related to a goal $g$, the relevant acceptability criteria might not allow much leeway compared to other metrics related to $g$ (e.g., it may happen that some goals with throughput values as different as 10 units from throughput values in $g$ are in the space of $g$, whereas none of the goal with latency values as different as 5 units from latency values in $g$ are within the space of $g$). Therefore, in general, the space of $g$ can be of arbitrary shape. To further develop the notion of configuration acceptability, it is useful to review concepts relating to relations and partial orders [16].

**Definition 2:** A *relation* $R$ on two sets $A$ and $B$ is a subset of the Cartesian product $A \times B$. A relation $R \subseteq A \times A$ is *reflexive* if $aRa$ for all $a \in A$. The relation $R$ is *symmetric* if $aRb$ implies $bRa$ for all $a, b \in A$. The relation $R$ is *anti-symmetric* if $aRb$ and $bRa$ imply $a = b$ for all $a, b \in A$. The relation $R$ is *transitive* if $aRb$ and $bRc$ imply $aRc$ for all $a, b, c \in A$.

**Definition 3:** A relation that is reflexive, anti-symmetric and transitive is a *partial order,* and we call a set on which a partial order is defined a *partially ordered set* [16]. For example, the relation “is a descendant of” is a partial order on the set of all people, if we view individuals as being their own descendants. As another example, the subset relation $'' \subseteq ''$ on all subsets of the set of integers is a partial order.

**Definition 4:** A partial order $R$ on a set $A$ is a *total order* if for all $a, b \varepsilon A$, we have $aRb$ or $bRa$ - that is, if every pairing of elements of $A$ can be related by $R$. For example the relation $'' \leq ''$ is a total order on the natural numbers [16].

In general, for a finite set $\Gamma$ of relevant goals, and a given acceptability relation, finding a minimal set of goals $\{g_1, g_2, \ldots, g_n\}$ such that $\bigcup_{i=1}^{n} space(g_i) = \Gamma$ is tractable and we will prove this in Section 6.2.3.1. The following result shows that the acceptability of configurations is a particularly well-behaved relation if it is a partial order.

**Theorem 1:** If we have a finite set $\Gamma$ of relevant goals, and the acceptability relation is a partial order, then there exists a unique, minimal set of goals $\{g_1, g_2, \ldots, g_n\}$ such that

$$\bigcup_{i=1}^{n} space(g_i) = \Gamma, \tag{26}$$

and this set of goals can be computed in polynomial time in $|\Gamma|$, the number of relevant goals.

**Proof:** Suppose there are two different minimal sets, $S_1$ and $S_2$, of goals that satisfy (26). Let $g_1$ be a goal such that $g_1 \in S_1$ and $g_1 \notin S_2$ and let $g_2$ be a goal such that $g_2 \in S_2$ and $g_2 \notin S_1$. Let $g \in S_2$ such that $g \to g_1$. The goal $g \notin S_1$ since $S_1$ is a minimal set. Thus, there exists a goal $g'$ such that $g' \in S_1$ and $g' \to g$. Since the acceptability relation is a partial order, $g'$ is acceptable for $g_1$ i.e., $g' \to g_1$. If $g' \neq g_1$, then it contradicts our assumption that $S_1$ is a minimal set. On the other hand, if $g' = g_1$, then $g' \to g \to g_1$ would imply $g' = g = g_1$ as the acceptability relation is a partial order, which would contradict the assumption that $g_1 \notin S_2$. Hence there is a unique minimal set that satisfies (26). Q.E.D.

If the acceptability relation is a partial order, the minimal set of goals is composed of only those goals that are not covered by any other goal in $\Gamma$. Therefore, this set of goals can be computed in polynomial time, using, for example, the following simple algorithm.

Construct a graph $Gr$ such that each goal in $\Gamma$ corresponds to a node in $Gr$. All the nodes that correspond to goals in the space of a goal $g$, where $g \in \Gamma$ are connected by directed edges from the node corresponding to goal $g$. For all the nodes, mark their immediate successors. The goals corresponding to the unmarked nodes constitute the above mentioned minimal set of goals. This set of unmarked nodes can be determined by a simple traversal of $Gr$.

**Example 7:** For a given application and the resource set, let $M = \{L, P, T\}$ be the set of relevant metrics, where $L$ denotes latency, $P$ denotes the average power, and $T$ denotes the average iteration period in the execution of the application. Consider the goals $g_1$ = [(*L*, 50), (*P*, 100), *T*], $g_2$ = [(*L*, 60), (*P*, 110), *T*], $g_3$ = [(*L*, 40), (*P*, 110), *T*)], and $g_4$ = [(*L*, 30), (*P*, 110), *T*]. The graph $Gr$ formed by nodes corresponding to these goals is shown in Figure 18, and each edge of the graph denotes that the goal corresponding to the source vertex is acceptable for the goal corresponding to the sink vertex. One can easily verify that in this case, the acceptability relation is a partial order, and the minimal set of goals that is needed to cover all four goals is $\{g_1, g_4\}$, which is unique. This is in accordance with Theorem 1. ■

**Definition 5:** *Dominance relation*: A point $p \varepsilon \Re^n$ dominates a point $\tilde{p} \varepsilon \Re^n$ if $p_i \le \tilde{p}_i$, for all $i = 1, \ldots, n$, where $p_i$ and $\tilde{p}_i$ denote $i$th components of $p$ and $\tilde{p}$ respectively.

**Example 8:** Transitive acceptability relation.

One can see that the dominance relation is a transitive relation. Also, the dominance relation is reflexive and anti-symmetric, which makes it a partial order. We can have an acceptability relation between goals based on the dominance relation where a goal $g_1$ is acceptable for a goal $g_2$ if and only if the constraint vector of the goal $g_1$ dominates the constraint vector of the goal $g_2$, and the residual objectives for both the goals are identical.

For a given application and resource set, let $M = \{L, P, T\}$ be the set of relevant metrics, where $L$ denotes the latency, $P$ denotes the average power, and $T$ denotes the average iteration period in the execution of the application. Consider the

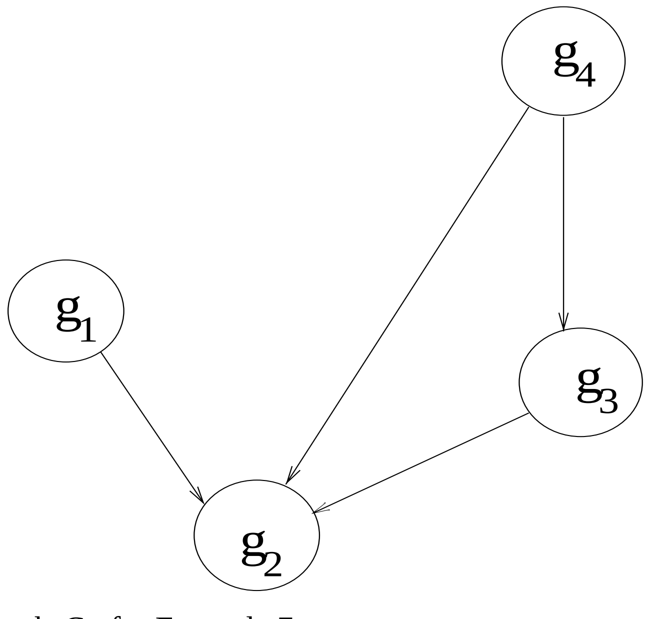


Figure 18. Graph $Gr$ for Example 7.

goals $g_1$ = [(*L*, 5), (*P*, 40), *T*], $g_2$ = [(*L*, 10), (*P*, 50), *T*], and $g_3$ = [(*L*, 10), (*P*, 60), *T*]. Using the acceptability condition based on the dominance relation defined above, we can see that $g_1 \to g_2$, $g_2 \to g_3$ and $g_1 \to g_3$ which verifies that the above relation is transitive. One can also verify that this acceptability condition is also reflexive and anti-symmetric. ■

**Definition 6:** A point $p \varepsilon \Re^n$ is a *Pareto point* for a mapping $\phi$: $\Re^n \to \Re^m$ if there exists no other point $\tilde{p} \varepsilon \Re^n$ such that

$$\phi_i(\tilde{p}) \leq \phi_i(p), \quad i = 1, \ldots, m, \tag{27}$$

where not all of the above inequalities are equalities [8]. The values $\phi_i(p)$ and $\phi_i(\tilde{p})$ denote the $i$th components of $\phi(p)$ and $\phi(\tilde{p})$, respectively. We can define Pareto dominance of a goal $g_1$ over a goal $g_2$ as the case when satisfying all constraints of $g_1$ leads to satisfying all constraints of $g_2$. It can be seen that if the residual objectives are same, the Pareto dominance relation is similar to acceptability based on the dominance relation, as defined earlier. One can observe that since Pareto dominance is a transitive relation, if the acceptability is based on Pareto dominance, then the acceptability relation also becomes a transitive relation and hence a partial order.

**Example 9:** Non-transitive acceptability relation.

Suppose that we have a single constraint metric, which is the average iteration period $T$ of the system. Thus, the constraint associated with a goal $g$ can be expressed as the desired average iteration period $T(g)$. Suppose that in a particular implementation context, the acceptability relation $g_1 \to g_2$ is defined by $T(g_1) - T(g_2) \leq \Delta T$ for some positive real number $\Delta T$. Thus, a configuration for $g_1$ can be worse than what is desired under $g_2$, and still acceptable for $g_2$, as long as the deviation does not exceed the threshold $\Delta T$. Assume that the goals $g_1, g_2$ and $g_3$ have desired average iteration period values of $T(g_1) = 5$, $T(g_2) = 5 - \frac{3\Delta T}{4}$ and $T(g_3) = 5 - \frac{3\Delta T}{2}$. One can see that $g_1 \to g_2$ and $g_2 \to g_3$ but $g_1$ is not acceptable for $g_3$. Therefore, this acceptability relation is not transitive.

To make the example more elaborate, assume that there is also a constraint associated with latency $L$, in addition to $T$. Let the acceptability relation $g_1 \to g_2$ now be defined by

$$T(g_1) - T(g_2) \leq \Delta T \text{ and } L(g_1) - L(g_2) \leq \Delta L. \tag{28}$$

Furthermore, let the desired average iteration period values be $T(g_1) = 5$, $T(g_2) = 5 - \frac{3\Delta T}{4}$, $T(g_3) = 5 - \frac{3\Delta T}{4}$, and let the desired latency values be $L(g_1) = 2$, $L(g_2) = 2 - \frac{3\Delta T}{4}$ and $L(g_3) = 2 - \frac{3\Delta T}{2}$. Once again, $g_1 \to g_2$ and $g_2 \to g_3$ but $g_1$ is not acceptable for $g_3$. Also, the graph induced by goals $g_1, g_2$ and $g_3$ is non-transitive (the graph is not identical to its transitive closure) and is shown in Figure 19, where vertices A, B and C represent goals $g_1$, $g_2$ and $g_3$ respectively. ■

**Example 10:** Transitive and symmetric acceptability relation.

A *cluster* of size $d$ is a maximal set $S$ of goals such that for every element $g \in S$, there exists at least one element $g' \in S$, such that none of the constraint values in $g'$ is different by more than $d$ units from the corresponding constraint value in $g$. Suppose that in a particular implementation context, the acceptability relation between two goals is defined in the following way.

*Two goals are acceptable for each other if and only if the constraint vectors of the two goals belong to the same cluster [37] of size 5 in the constraint vector space.* It can be seen that this acceptability relation is transitive, as if $g_1 \rightarrow g_2$ and $g_2 \rightarrow g_3$, then all of $g_1$, $g_2$ and $g_3$ belong to the same cluster, and hence $g_1 \rightarrow g_3$. One can also observe that this acceptability relation is also symmetric because if $g_1 \rightarrow g_2$, then $g_2 \rightarrow g_1$. ■

### 6.2.2 On-line management and use of configurations

If we have an acceptability relation between goals based on the dominance relation, then results related to partial orders can be applied to the management of goals in an associated configuration store. This leads to valuable properties such as that exposed by Theorem 1. Also, the dominance relation is a very natural candidate for an acceptability relation among goals, as a configuration corresponding to the dominating goal can be used in place of a configuration corresponding to the dominated goal without violating any constraints. This all motivates our use of the domi-

$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

Figure 19. Induced graph.

nance relation in managing configuration stores. The following is a framework for on-line management of goals and configurations, where if a suitable configuration for the applied goal is not found in the configuration store then a configuration suitable for a *weaker* goal (one that is dominated by the applied goal) is looked for to be used, and so on, until one finds a suitable configuration. The following meta-algorithm forms the core of the on-line refinement block in Figure 17. This core component deals with the management of goals and configurations. This is a meta-algorithm because specific details of the architecture, the application, and the on-line adaptation algorithms are left unspecified, and can be customized based on the relevant classes of applications and architectures.

This meta-algorithm consists of the following steps.

• Initialize the variable *currentObjective* to be the residual objective.

• If there exist one or more configurations in the configuration store whose constraints dominate the constraints of the given goal $g$, then select the one that best addresses the current objective, and continue optimizing the current objective through *on-line adaptation* (while $g$ is in effect).

• Otherwise, discard the current objective, demote the lowest priority constraint to be the new current objective (call this a *demoted constraint objective,* abbreviated *DMC*), and repeat Step 1.

• During on-line adaptation, if we are working on improving a DMC, and we achieve a configuration that satisfies the associated constraint (without violating any higher-priority constraints), then promote the DMC back to being a (regular) constraint, and (if applicable) move to the next (lower priority) constraint (which must be a discarded DMC), and promote this to be the current objective. If there are no more constraints, then set the current objective to be the residual objective.

The detailed pseudocode for the above algorithm framework is shown in Figure 20. This meta-algorithm maintains a *current objective* at all times, where the

```
**Global variables**: goal $g$, $g_c$, $g_o$; Stack $S$; global clock time $t$; time $timelimit$
**Function** onLineManagement

$g_c = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$ /*Current goal */
$g = g_c = g_o$
Instantiate a stack $S$. Initialize $S$ to an empty stack.
$objective_c = m_R$
$constraint_c = \text{null}$
while there does not exist a $C \varepsilon$ Configuration store, such that
constraints of $C$ dominate the constraints of $g$ {
      ($g$, $constraint_c$, $objective_c$) = demoteConstraint($g$, $S$)
}
Find all $C \varepsilon$ Configuration store, such that constraints of $C$
dominate the constraints of $g$. Select the one that
addresses the $objective_c$ best and store it in $configuration_c$.
while (($t < timelimit$) & ($g_c == g_o$)){
      while $constraint_c$ is not satisfied {
            onLineAdaptation($g$, $objective_c$, $configuration_c$)
      }
      ($g$, $constraint_c$, $objective_c$) = promoteConstraint($g$, $S$)
}
```

Figure 20. Pseudocode for on-line configuration management.

goal is always to improve the current objective without violating any of the previously satisfied constraints. This current objective is changed to improving the metric associated with the next unsatisfied constraint once the constraint associated with the current objective is satisfied. The function *onLineAdaptation* takes a goal, objective metric, and configuration as inputs, and keeps refining the configuration in an effort to continually improve its quality (as defined by (25)). The value $g_c$ represents the goal that has been applied to the system. In the pseudocode, it is assumed that $g_c$ will change to the new goal vector when the goal vector applied to the system changes. Pseudocode for the related functions is given in Figure 21. The on-line refinement part of this configuration management framework, which continually refines a configuration after it is picked from the configuration store, has been implemented and some experimental results pertaining to it are discussed in Section 6.3.

### 6.2.3 Models for problems related to configuration management

Here we study some fundamental versions of the problems related to the configuration management, improve our understanding of their complexity, and relate aspects of them to well-studied problems. For these purposes, it is helpful to define a notion of "distance" (e.g., Euclidean distance in goal space) between two goals. If the distance between a goal $g_1$ and another goal $g$ is less than the distance between a third goal $g_2$ and the goal $g$, then one may say that goal $g$ is *closer* to goal $g_1$ than $g_2$. One would like to have configurations in the configuration store such that whole of range of goals is covered, which means that for any goal $g$, there is at least one goal $g'$ and its configuration $C$ in the configuration store such that the space of $g'$ includes $g$. There are two related problems regarding the size of the configuration store.

**P1.** Find the minimum size configuration store and the goals that should be stored in it such that all the relevant goals are covered and,

**P2.** given a fixed size configuration store, find the goals whose configurations should be stored such that the sum of the distances of those goals that are not present in configuration store, from the closest goal present in configuration store, is

```
Function promoteConstraint
Input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$, Stack $S$
Output goal $g'$
constraint value $v = S$.pop()
metric $m = S$.pop()
constraint value $x = S$.pop()
$S$.push($x$)
Instantiate $g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), (m_K, v), m]$
Output {$g'$, $x$, $m$ )

Function demoteConstraint
Input goal $g = [(m_1, c_1), (m_2, c_2), \ldots, (m_K, c_K), m_R]$, Stack $S$
Output goal $g'$
$S$.push($m_R$)
$S$.push($c_K$)
Instantiate a new goal
$g' = [(m_1, c_1), (m_2, c_2), \ldots, (m_{K-1}, c_{K-1}), m_K]$.
Output ($g'$, $c_K$, $m_K$ )
```

Figure 21. Pseudo-code for functions `promoteConstraint` and `demoteConstraint` from Figure 20.

minimum. For this, one needs to have a well-defined measure of distance between goals.

We explain these problems in detail and reduce these to some well-studied problems.

#### 6.2.3.1 Analysis of the configuration management problem P1

The concept of a dominating set is useful in understanding P1.

**Definition 7:** For a directed graph $G(V, E)$, a subset $D$ of $V$ is a *dominating set* if for all $v \varepsilon V$, either $v \varepsilon D$ or there exists $u \varepsilon D$, such that $(u, v) \varepsilon E$.

**Definition 8:** *Minimum dominating set problem*: Given a directed graph, find a minimum dominating set of the graph.

**Example 11:** For the graph given in Figure 22, the dominating sets are (A,B), (A,C), (B,C) and (A,B,C). Minimum dominating sets for the graph are (A,B), (A,C) and (B,C). ■

We can model problem P1 as one of the graph problems in the following

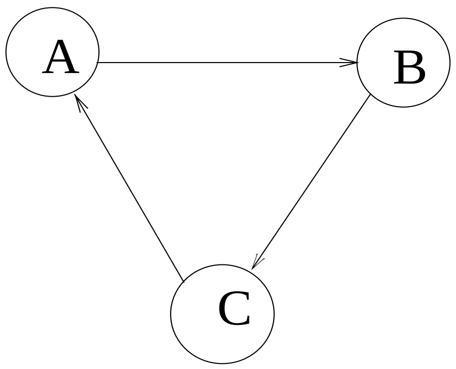


Figure 22. Illustration for the dominating set of a graph.

way. Construct a directed graph $G(V, E)$ in the following manner. Initialize $V$ to be a set of vertices such that each vertex in $V$ corresponds to a goal $g \in \Gamma$, where $\Gamma$ denotes the set of relevant goals. For each of the vertices $v \varepsilon V$, where $v$ corresponds to some goal $g$, construct directed edges $(v, u)$ belonging to $E$ such that $u$ belongs to the set of vertices corresponding to the space of $g$. Find the minimum dominating set [16] of the graph $G$. This problem is NP-complete, but polynomial time 2-approximation algorithms exist [17]. The set of vertices thus obtained would correspond to the set of goals whose configurations need to be stored.

To reduce P1 from the minimum dominating set problem, for every vertex in the dominating set problem, instantiate a goal; and for every edge, instantiate a condition that the goal corresponding to the source vertex is acceptable for the goal corresponding to the sink vertex. The problem P1 related to this set of goals and acceptability relation among goals is equivalent to the given minimum dominating set problem instance. The vertices in the given minimum dominating set problem instance, corresponding to the goals that should be stored in the configuration store, found by solving P1, constitute a minimum dominating set for the given minimum dominating set problem instance. This proves that the problem P1 is NP-hard. Problem P1 can be posed as a decision problem in the following manner.

*Is a configuration store of size $x$ and a set $S$ of goals stored in it sufficient for covering all the goals?*

A solution to this problem can be verified by checking if the set $S$ of goals covers all the relevant goals, and checking if the set $S$ can be stored in a configuration store of size $x$. It can be done in polynomial time in terms of the number of goals by checking for each goal if that goal or any goal that covers it is in the set $S$. So, problem P1 is NP-complete.

**Example 12:** The instance of Problem P1 that is obtained by reduction from the minimum dominating set instance of Example 11 has three instantiated goals $g_1$, $g_2$ and $g_3$ corresponding to vertices $A$, $B$ and $C$, respectively. The diagram in Figure 23 shows the instantiated goals. We have, $g_1 \rightarrow g_2$, $g_2 \rightarrow g_3$ and $g_3 \rightarrow g_1$. The smallest sets of goals that cover all the three goals are, $\{g_1, g_2\}$, $\{g_2, g_3\}$, and $\{g_3, g_1\}$, which is in accordance with the minimum dominating sets (A,B), (A,C) and (B,C), in Example 11. ■

If the acceptability relation is a partial order, then the minimum dominating set can be found in polynomial time by picking up all the vertices with no incoming edges in the dominating set. This is in accordance with Theorem 1. The goals corresponding to these vertices would give the minimal set of goals that covers all relevant goals.

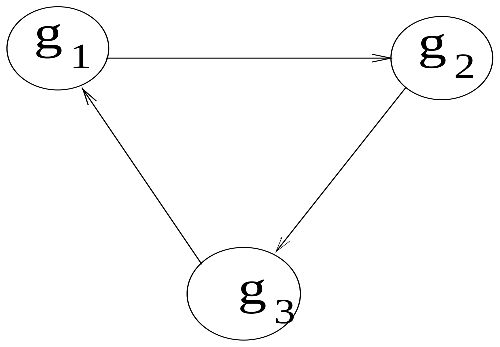


Figure 23. Illustration for Example 12.

The above formulation does not differentiate between cases when the goals that are being covered are at closer distances from the goals covering them than cases where they are at larger distances unless the size of the solution found is not same.

### 6.2.3.2 Analysis of configuration management problem P2

We can model problem P2 as a well-defined graph-theoretic problem in the following way. Given a fixed size configuration store, there is a fixed number of configurations that can be stored in it. Let this number be $k$. Consider the problem of computing goals whose configurations should be stored such that the sum of the distances of those goals that are not present in the configuration store from the closest goal present in the configuration store is minimum. This problem can be directly related to one of the well-studied problem, called *$k$-median problem* [20]. We assume that there is a pre-defined way of computing distance between any two goals and we are given the distances between any two goals in the goal space. Also, we are assuming that our goal space is a metric space, which has been defined in Section 1.1.

**Definition 9:** *k-median problem*: In the *k*-median problem, we are given a set of potential facility locations $F$. Any open facility can provide an unlimited amount of a certain commodity. There is a set of clients or demand points $D$ that require service; client $j \varepsilon D$ has a positive demand of commodity $d_j$ that must be shipped from one of the open facilities. If a facility at location $i \varepsilon F$ is used to satisfy the demand of client $j \varepsilon D$, the service or transportation cost incurred is proportional to the distance $c_{ij}$ from $i$ to $j$. This distance function $c$ is non-negative, symmetric and satisfies the triangle inequality. The goal is to determine $k$ potential facility locations at which to open facilities and an assignment of clients to these facilities so as to minimize the overall service cost.

Problem P2 can be modeled as a $k$-median problem in the following way. For every goal present in the goal space, instantiate a location in $n$-dimensional space where $n$ is the dimension of goal space. The distance between any two instantiated locations in this $n$-dimensional space is equal to the distance between the corresponding goals in the goal space. Instantiate a set of possible facility locations and initialize it to the set of all instantiated locations in the $n$-dimensional space. Instantiate a set of all client locations and initialize it to the set of all instantiated locations in the $n$-dimensional space.

The $n$-dimensional space, the set of possible facility locations, the set of all client locations and the distances between any two locations in the $n$-dimensional space constitute an instance of the $k$-median problem. Problem P2 can be solved by modeling it as a $k$-median problem, as described above, and then solving that $k$-median problem. The goals corresponding to the set of locations at which facilities should be opened, found by solving the associated $k$-median problem, would be the answer to the problem P2.

For the simple case of two-dimensional space, polynomial-time approximation algorithm with a 3-approximation factors exist [12] for $k$-median problem.

**Example 13:** Consider a goal space composed of three goals $g_1$, $g_2$ and $g_3$, as shown in Figure 24. The numbers beside the edges connecting the goals represent the distances between the goals. One can see that the triangle inequality holds. Let the size of the fixed size configuration store be 1 unit (i.e., the configuration store can store information for one goal). Problem P2 for this specific case is

*Find one goal, out of $g_1$, $g_2$ and $g_3$, that should be stored in the configuration store such that the sum of its distances from the other two goals is minimum.*

The corresponding $k$-median problem obtained after modeling this problem based on the concepts above is shown in Figure 25. Locations A, B and C corre-

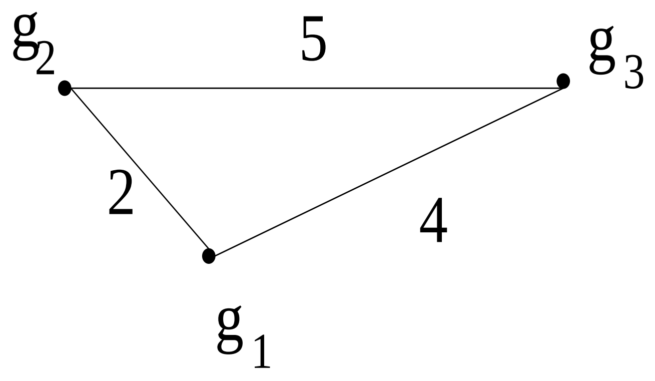


Figure 24. An example goal space.

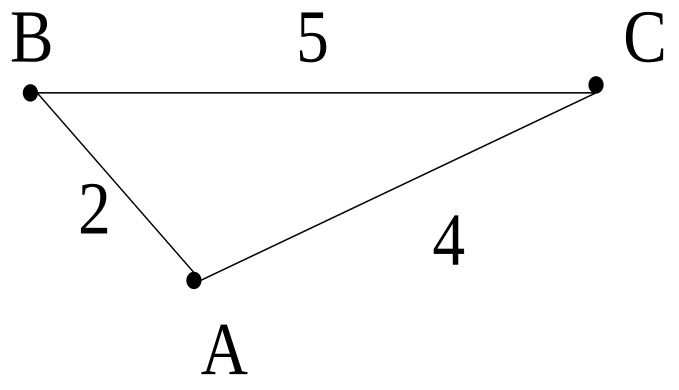


Figure 25. Illustration for Example 13.

spond to goals $g_1$, $g_2$ and $g_3$, respectively. The set of possible facility locations is {A,B,C}, and the set of all client locations is also {A,B,C}. The distances between all locations are represented by numbers beside the edges in the figure. Solving the $k$-median problem for $k = 1$ would give A as the location where the facility should be opened. This would correspond to storing goal $g_1$ in the configuration store as an answer to problem P2. ■

There can be several other variants of problem P2. For example, one of the variants is to find the goals whose configurations should be stored such that least number of goals in the goal space are left uncovered.

#### 6.2.3.3 Exploring trade-offs

Configuration management problems P1 and P2 can be viewed as extreme in the sense that in one of them we want to cover all feasible goals without considering how large the minimum size configuration store would be (P1), and in the other case, we have a fixed size configuration store and we are trying to find out the maximum number of goals that can be covered using that configuration store even though that number could be much less than the total number of relevant goals (P2). A more elaborate formulation would be one in which we have to pay extra cost for increasing the size of configuration store, but we would be gaining some additional service by that (e.g., now some goals that are not present in the configuration store are closer to the goals that are present in the configuration store). This way we can explore various trade-offs between the size of the configuration store vs. the number of goals stored in a well-defined way. Understanding the so-called facility location problem would be useful in this regard.

**Definition 10:** *Facility location problem* [15]: In the facility location problem, we are given a set of potential facility locations $F$; building a facility at location $i \varepsilon F$ has an associated nonnegative fixed cost $f_i$, and any open facility can provide an unlimited amount of certain commodity. There is a set of clients or demand points $D$ that require service; client $j \varepsilon D$ has a positive demand of commodity $d_j$ that must be shipped from one of the open facilities. If a facility at location $i \varepsilon F$ is used to satisfy the demand of client $j \varepsilon D$, the service or transportation cost incurred is proportional to the distance $c_{ij}$ from $i$ to $j$. The distance function $c$ is non-negative, symmetric and satisfies the triangle inequality. The goal is to determine a subset of the set of potential facility locations at which to open facilities and an assignment of clients to these facilities so as to minimize the overall total cost.

We define a new problem P3 so that the trade-offs between the size of the configuration store vs. the number of goals stored can be explored by modeling P3 as a facility location problem. We assume that there is a defined way of computing distance between any two goals and we are given the distances between any two goals in the goal space. Also, we are assuming that our goal space is a metric space, which is defined in Section 1.1.

**Problem P3**: Find the size of the configuration store and the goals that should be stored in it such that the overall cost is minimum. The overall cost is equal to the sum of the distances of the goals that are not stored in the configuration store, from the closest goal that is stored in the configuration store plus the cost associated with storing goals in the configuration store.

Problem P3 can be modeled as a facility location problem in the following way. Let $w_i$ be the cost associated with storing a configuration-goal pair associated with goal $g_i$ in the configuration store. For every goal present in the goal space, instantiate a location in $n$-dimensional space where $n$ is the dimension of goal space. The distance between any two instantiated locations in this $n$-dimensional space is equal to the distance between the corresponding goals in the goal space. Instantiate a set of possible facility locations and initialize it to the set of all instantiated locations in the $n$-dimensional space. Instantiate a set of all client locations and initialize it to the set of all instantiated locations in the $n$-dimensional space.

The $n$-dimensional space, the set of possible facility locations, the set of all client locations, the cost $w_i$ associated with storing any goal $g_i$ in the configuration store, and the distances between any two locations in the $n$-dimensional space constitutes an instance of the facility location problem. Problem P3 can be solved by modeling it as a facility location problem, as described above, and then solving that facility location problem. The goals corresponding to the set of locations at which facilities should be opened and the size of the configuration store corresponding to the cost of building those facilities, found by solving the associated facility location problem, would solve an associated instance of problem P3.

**Example 14:** Consider the same goal space we used in Example 13, which was composed of three goals $g_1$, $g_2$ and $g_3$ and is shown in Figure 24. The numbers beside the edges connecting the goals represent the distances between the goals. One can see that the triangle inequality holds. Let the cost of storing a goal in the configuration store be the same for all the goals and be equal to 3 units. Problem P3 for this specific case is

*Find the subset of* $\{g_1, g_2, g_3\}$ *that should be stored in the configuration store such that the overall cost is minimum. This overall cost is equal to the sum of the distances of the goals that are not stored in configuration store from the closest goals present in the configuration store plus the total cost of storing the goals that are present in the configuration store.*

The corresponding facility location instance obtained after modeling this P3 instance as a facility location problem is shown in Figure 26. Locations A, B and C correspond to goals $g_1$, $g_2$ and $g_3$, respectively. The set of possible facility locations is {A,B,C}, and the set of all client locations is also {A,B,C}. The distances between all locations are represented by numbers beside the edges in the figure. Solving the facility location problem would give {A, C} or {B, C} be the set of loca-

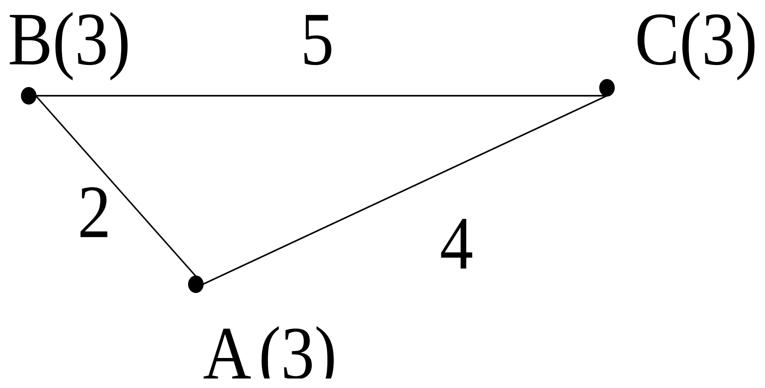


Figure 26. Illustration for Example 14.

tions where the facilities should be opened. The cost associated with any of the above answers is 8 units and is the minimum cost for this example. This corresponds to the optimum solution of a configuration store of size 2 and set $\{g_1, g_3\}$ or $\{g_2, g_3\}$ to be the set of goals that are stored in the configuration store. ■

Polynomial-time algorithms with an approximation guarantee of 1.74 exist for the facility location problem [15].

When the acceptability relation is a partial order, the formulations in Section 6.2.3.2 and Section 6.2.3.3 have no known polynomial-time solutions unlike formulation in Section 6.2.3.1. However, these formulations are nevertheless important as they isolate specific concerns in the design of configuration store.

## 6.3. On-line algorithms

In this section, we will discuss low-complexity online refinement algorithms that are represented by function *onLineAdaptation* in Figure 20. These online algorithms are responsible for the refinement of a configuration that is picked from the configuration store so that it better matches the applied goal. This is performed according to the configuration management framework discussed in Section 6.2.2.

### 6.3.1 Simple low-complexity, run-time refinement approach

Here we develop low-complexity, on-line strategies based on heuristics for throughput and power optimization. Since these algorithms are employed during run-time, they must be of low-complexity. Also, the algorithms developed should be able to handle stochastic application behavior, and must be dynamic in structure. Our configuration management framework provides us with an initial configuration, which is refined by these on-line strategies for the objective specified by the configuration management framework as shown in Figure 20. In this framework in Figure 20, these on-line strategies are represented by function *onLineAdaptation.* These strategies are applied while the application executes as otherwise they will take away too much computational resources away from the application itself. Also, it is necessary to apply these on-line strategies as the application executes because execution of the application is necessary to generate statistics about the application, which in turn are useful in driving the reconfiguration process.

The approach of taking feedback from the execution of the application makes these on-line methods able to handle even applications with stochastic attributes that have *slowly-varying* distributions, in addition to applications with fixed execution times, and applications with stochastic attributes that have stationary distributions. In general, the online refinement problem can thus be viewed as a problem of tracking the dynamics of the goal and the characteristics of the application.

### 6.3.2 Throughput optimization

Load balancing algorithms are generally designed to equally spread the load on processors and maximize their utilization while minimizing the total task execution time [42]. A dynamic load balancing mechanism has to allocate tasks to the processors dynamically as they arrive. As redistribution of tasks has to take place during run-time, dynamic load balancing mechanisms are usually harder to implement. To demonstrate the ability of our online configuration management framework, of Figure 20, we used a simple heuristic based on load balancing to optimize throughput, which is as follows.

Maximally and minimally loaded processors are identified on the basis of past execution statistics. Then a task is chosen randomly from a maximally loaded processor and scheduled at the earliest available place on a minimally loaded processor. This method helps balance loads over processors. If the new schedule thus formed performs better then the older schedule, then it becomes the schedule with which the system continues, and is further refined in the same way to yield better schedules. If this new schedule performs worse than the older schedule then in the older schedule, some other task is moved from the maximum loaded processor to minimum loaded processor and the performance of the schedule is observed in the same way as above. This way, we keep refining the schedule so as to optimize throughput. Pseudocode for this heuristic is represented by function *onLineAdaptationTr* and is given in Figure 27. In the pseudocode, *moveTaskTr*$(s, n)$ is a function that chooses $n$-tasks from the maximum loaded processor in a schedule $s$ randomly, moves them to appropriate locations on the minimum loaded processor, and returns the modified schedule. Randomization in choosing tasks from the maximum loaded processor provides a low-complexity approach to increase the explored region of the design space. The function *executeTr*$(s, l)$ is a function that executes the application according to schedule $s$ for an interval of time length $l$ and returns the throughput of the application during that interval. The value of $l$ to use, depends on non-determinacy of the application (11). Generally, the more non-deterministic the application is, the longer it needs to be executed, so that more accurate value of average throughput can be calculated.

The function *onLineAdaptationTr* returns a schedule that it deems most appropriate for throughput maximization. Note that if moving any single tasks from the maximum loaded processor to the minimum loaded processor does not improve

```
Function: onLineAdaptationTr
Input: Schedule $s_i$, time $timelimit$, time $l$
Output: Schedule $s$

$t_{old}$ = executeTr($s_i$, $l$)
$s_{old}$ = $s_i$
$n$ = 1
while (clock < $timelimit$) {
      $s$ = moveTaskTr($s_{old}$, $n$)
      if (exhausted all $n$-tasks movements and still no
       improvement){
            $n$ = $n + 1$
            $s$ = moveTaskTr($s_{old}$, $n$)
      }
      $t$ = executeTr(($s$, $l$)
      if($t \geq t_{old}$) {
            $s_{old}$ = $s$
            $t_{old}$ = $t$
            $n$ = 1
      }
      clock = clock + $l$
}
return $s_{old}$
```

Figure 27. Pseudo-code for throughput optimization.

performance then the heuristic chooses a pair of tasks to be moved to other processor. This approach of progressively increasing the number of tasks to be moved continues whenever all combinations for a particular number of tasks have been exhausted. This approach thus attempts to make small low-complexity changes first and if that does not improve performance, the approach gradually reaches towards higher-complexity changes. This approach of starting off with lower complexity changes is followed all through the algorithm whenever a change in the schedule is needed. The higher complexity changes are larger in number than small low-complexity changes, and help the system in escaping from local minima.

### 6.3.3 Power optimization

There are several factors that affect the power consumption in the class of architectures under investigation. These factors include inter processor communication (IPC), the assignment of supply voltage levels to processors (if voltage scaling is an option), etc. To find a configuration that reduces the power consumption, we use an approach very similar to the one developed for throughput optimization in Section 6.3.2. While minimizing power consumption, we try to minimize IPC cost, as IPC typically consumes significantly more power than normal task execution. This is achieved by selectively moving tasks from the maximally loaded processor to the minimally loaded one, and by choosing the tasks to be moved on the basis of the IPC associated with them. The higher the IPC associated with a task, the higher its chances are of being transferred to another processor. Pseudocode for such on-line, power-optimized scheduling is shown in Figure 28. The function *onLineAdap-*

*tationPower* returns a schedule that it deems most appropriate for power minimization. In the pseudocode, *moveTaskPower*($s$, $n$) is a function that chooses $n$-tasks from the maximally loaded processor in schedule $s$ on the basis of the net IPC associated with them, and moves them to the minimally loaded processor, and returns the modified schedule. This net IPC for an actor is calculated by adding all the IPC costs in the system in which the actor is either a source actor or a sink actor. The

```
Function: onLineAdaptationPower
Input: Schedule $s_i$, time $timelimit$, time $l$
Output: Schedule $s$

$P_{old}$ = executePower($s_i$, $l$)
$s_{old}$ = $s_i$
$n$ = 1
while (clock < $timelimit$) {
        $s$ = moveTaskPower($s_{old}$, $n$)
        if (exhausted all $n$-tasks movements and still no
         improvement){
                $n$ = $n + 1$
                $s$ = moveTaskPower($s_{old}$, $n$)
        }
        $P$ = executePower(($s$, $l$)
        if($P \le P_{old}$) {
                $s_{old}$ = $s$
                $P_{old}$ = $P$
                $n$ = 1
        }
        clock = clock + $l$
}
return $s_{old}$
```

Figure 28. Pseudo-code for power optimization.

function *executePower*($s$, $l$) is a function that executes the application according to the given schedule $s$ for an interval of time length $l$ and returns the power consumption during that interval. Again a progressive approach similar to the on-line throughput optimization algorithm is followed that tries out small changes first and moves on to larger changes only when there is no further improvement.

One major difference is that the tasks to be moved are chosen from the maximally loaded processor on the basis of the IPC associated with them unlike the case with throughput optimization, where tasks are chosen randomly from the maximum loaded processor.

### 6.3.4 Experimental results

Table 5 shows the results of applying the on-line throughput-optimization algorithm, described in Figure 27 in conjunction with the general online configuration management framework of Figure 20, to various DSP benchmarks. For a non-negative integer $k$, $T_k$ represents the average iteration period of the best schedule found by the throughput-optimization algorithm after $k$ schedules have been executed for a fixed amount of time each, in order to assess the throughput associated with them. By the same token, $T_0$ represents the average iteration period associated with the starting schedule, which is found by using standard *critical path scheduling* [37]. The critical path length is computed in terms of average execution times of actors. It is observed that irrespective of the degree of non-determinacy $\lambda$ of the application, this approach finds a schedule during run-time that outperforms a schedule generated using critical path scheduling.

Similarly, Table 6 shows the results of applying the power-optimization algorithm, described in Figure 28, to various DSP benchmarks. For a non-negative integer $k$, $P_k$ represents the average power consumed per iteration of the best schedule found by power optimization algorithm after $k$ schedules have been executed for a fixed amount of time each in order to assess the power consumption associated with them. The starting schedule is found, as with the throughput optimization experiments, by using the standard critical path scheduling algorithm. Interprocessor communication (IPC) per time unit during the execution is taken as an estimate for power consumption. Since IPC consumes significant amounts of power,

| Appli cation | $\lambda$ | $T_0$ | $T_{10}$ | $T_{20}$ | $T_{30}$ | $T_{40}$ | $T_{50}$ |
|---|---|---|---|---|---|---|---|
| fft1 | 0 | 273 | 273 | 269 | 269 | 268 | 243 |
| fft1 | 276 | 276 | 276 | 276 | 270 | 269 | 256 |
| fft1 | .359 | 306 | 280 | 280 | 280 | 280 | 280 |
| qmf | 0 | 145 | 145 | 145 | 142 | 140 | 140 |
| qmf | .256 | 156 | 145 | 145 | 142 | 142 | 142 |
| qmf | .568 | 151 | 151 | 151 | 131 | 131 | 131 |
| karp | 0 | 395 | 376 | 375 | 375 | 375 | 375 |
| karp | .309 | 421 | 389 | 389 | 362 | 341 | 277 |
| karp | .608 | 475 | 411 | 411 | 408 | 405 | 403 |
| meas | 0 | 220 | 187 | 187 | 187 | 187 | 185 |
| meas | .207 | 232 | 206 | 188 | 188 | 188 | 188 |
| meas | .405 | 247 | 202 | 196 | 196 | 196 | 196 |

Table 5. Result of applying algorithm in Figure 27 on various DSP benchmarks.

and all processors are assumed to be homogeneous, and accordingly, are approximated as consuming equal amounts of power, to a first degree of approximation, the overall level of power consumption can be estimated as IPC per unit time. It is observed that the power-optimization algorithm also is able to find schedules with better performance than the schedules generated by standard critical path scheduling.

The on-line adaptation framework for refining a given goal is shown in Figure 29 and it is a part of the on-line configuration management framework shown in

| Appli cation | $\lambda$ | $P_0$ | $P_{10}$ | $P_{20}$ | $P_{30}$ | $P_{40}$ | $P_{50}$ |
|---|---|---|---|---|---|---|---|
| fft1 | 0 | .274 | .245 | .201 | .172 | .168 | .151 |
| fft1 | .117 | .268 | .200 | .179 | .179 | .179 | .179 |
| fft1 | .359 | .257 | .173 | .173 | .173 | .173 | .173 |
| qmf | 0 | .133 | .111 | .105 | .099 | .060 | .060 |
| qmf | .256 | .123 | .115 | .103 | .080 | .073 | .068 |
| qmf | .568 | .128 | .120. | .076 | .076 | .076 | .076 |
| karp | 0 | .131 | .131 | .073 | .073 | .061 | .061 |
| karp | .309 | .122 | .122 | .075 | .072 | .065 | .065 |
| karp | .608 | .132 | .131 | .098 | .096 | .095 | .086 |
| meas | 0 | .054 | .050 | .026 | .026 | .026 | .024 |
| meas | .207 | .054 | .054 | .054 | .018 | .018 | .009 |
| meas | .405 | .059 | .059 | .055 | .040 | .021 | .007 |

Table 6. Results of applying power optimization algorithm of Figure 28 to various DSP benchmarks.

Figure 20. In the on-line adaptation framework shown in Figure 29, depending upon $objective_c$, one of the available on-line functions, such as *onLineAdaptationTr* or *onLineAdaptationPower*, would be chosen, for on-line adaptation of the given configuration for the applied goal. Other details of on-line adaptation are explained in Section 6.2.

Table 7 shows the performance of our on-line adaptation framework, together with the *onLineAdaptationTr* and *onLineAdaptationPower* functions, for various goals applied to several DSP benchmarks. The starting schedule that is refined is found, as with the other experiments in this section, by using the standard critical path scheduling algorithm. The set of relevant metrics $M$ for our experiments is $M = \{T, P\}$, where $T$ denotes the average iteration period of the execution and $P$ denotes the average power consumption. Experiments are reported for the following eight goals.

```
Input $objective_c$, $constraint_c$, $configuration_c$, $timelimit$, $g$, $g_c$,
      $g_o$
Output A refined schedule

while (($t < timelimit$) & ($g_c == g_o$)){
      while $constraint_c$ is not satisfied {
            onLineAdaptation($g$, $objective_c$, $configuration_c$)
      }
      ($g$, $constraint_c$, $objective_c$) = promoteConstraint($g$, $S$)
}
```

Figure 29. On-line adaptation framework.

$g_1$ = {(P, 0.270), (T, 265), (P, 0.250), (T, 0.255), P}

$g_2$ = {(T, 260), (P, 0.240), T}

$g_3$ = {P, 0.125), (T, 180), P}

$g_4$ = {(T, 165), (P, 0.110), (T, 160), P}

$g_5$ = {(T, 360), (P, 0.160), (T, 355), (P, 0.155), (T, 350) P}

$g_6$ = {(T, 345), P}

$g_7$ = {(T, 215), (P, 0.040) T}

$g_8$ = {(P, 0.053), (T, 215), (P, 0.050), (T, 210), P}

In Table 7, the column titled “Goal” represents the goal that is applied to the application. Also, for a non-negative integer $k$, column $v_k$ denotes the value of a metric of the best schedule found by the on-line adaptation framework, after $k$ schedules have been assessed by executing them for some time. For the same experiments, which are reported in Table 7, Table 8 shows the times at which different constraints associated with the applied goals, are satisfied. For a given goal that is applied on an application, for non-negative integers $n$ and $i$, $n_i$ denotes the number of schedules that have been executed in order to assess them before the $i$th constraint in the applied goal is satisfied. One can see that our on-line framework is able to meet the constraints specified in the goal within a reasonable number of schedules. This is in accordance with the effectiveness of our on-line algorithms as demonstrated in experimental results earlier in this section.

# 7. Conclusion and future work

In this report, we have defined a new measure of latency, called periodic-output latency, which is more appropriate than conventional measures, for self-timed DSP systems. We have developed of several techniques to reduce the transient and

| Appli cation | λ | Goal | Metr ic | $v_0$ | $v_{10}$ | $v_{20}$ | $v_{30}$ | $v_{40}$ | $v_{50}$ | $v_{60}$ |
|---|---|---|---|---|---|---|---|---|---|---|
| fft1 | 0 | $g_1$ | T | 278 | 278 | 278 | 278 | 256 | 254 | 254 |
| | | | P | .273 | .269 | .269 | .269 | .204 | .226 | .226 |
| fft1 | .359 | $g_2$ | T | 309 | 256 | 251 | 251 | 251 | 252 | 259 |
| | | | P | .242 | .282 | .278 | .278 | .278 | .257 | .221 |
| qmf | 0 | $g_3$ | T | 145 | 242 | 198 | 198 | 186 | 170 | 170 |
| | | | P | .133 | .117 | .098 | .098 | .088 | .096 | .096 |
| qmf | .256 | $g_4$ | T | 142 | 164 | 162 | 162 | 153 | 153 | 153 |
| | | | P | .136 | .127 | .110 | .110 | .110 | .110 | .110 |
| karp | 0 | $g_5$ | T | 395 | 353 | 346 | 342 | 342 | 342 | 342 |
| | | | P | .131 | .158 | .156 | 148 | .148 | .148 | .148 |
| karp | .309 | $g_6$ | T | 450 | 352 | 300 | 342 | 342 | 346 | 346 |
| | | | P | .115 | .155 | .159 | .151 | .151 | .148 | .148 |
| meas | 0 | $g_7$ | T | 220 | 212 | 201 | 184 | 184 | 184 | 184 |
| | | | P | .054 | .075 | .059 | .021 | .021 | .021 | .021 |
| meas | .405 | $g_8$ | T | 185 | 218 | 212 | 212 | 212 | 210 | 196 |
| | | | P | .064 | .018 | .037 | .037 | .037 | .019 | .040 |

Table 7. Results for on-line framework tracking an applied goal.

periodic-output latency in self-timed systems. We have also proposed a way of streamlining our scheduling strategies for deterministic, contention -free self-timed systems using a graph-theoretic framework.

It is observed that, in general, there is an improvement in the transient by using the transient-reduction scheme for self-timed systems with deterministic execution times. Also, the latency-reduction scheme reduces latency significantly for both deterministic and non-deterministic systems. For the algorithm to solve the TBL problem, there is some scope for improvement by using a more appropriate algorithm in the first stage. This has been discussed in the following paragraphs.

Hoang's algorithm is greedy in nature. It tries to exploit the parallelism in a particular stage to the fullest by assigning actors to different processors if it improves throughput. In this process, it may run out of processors and will not be

| Appli cation | $\lambda$ | Goal | $n_1$ | $n_2$ | $n_3$ | $n_4$ | $n_5$ |
|---|---|---|---|---|---|---|---|
| fft1 | 0 | $g_1$ | 1 | 37 | 39 | 42 | - |
| fft1 | .359 | $g_2$ | 7 | 56 | - | - | - |
| qmf | 0 | $g_3$ | 8 | 48 | - | - | - |
| qmf | .256 | $g_4$ | 0 | 13 | 36 | - | - |
| karp | 0 | $g_5$ | 4 | 7 | 9 | 28 | 28 |
| karp | .309 | $g_6$ | 16 | - | - | - | - |
| meas | 0 | $g_7$ | 8 | 28 | - | - | - |
| meas | .405 | $g_8$ | 3 | 17 | 17 | 48 | - |

Table 8. Results for on-line framework tracking an applied goal.

able to provide any schedule that meets the throughput requirement even though one may exist. The constraint of not allowing any pipeline stage more than $T$ time to execute, is more than necessary for the case of self-timed execution as one may have pipeline stages such that the time taken to execute a pipeline stage is greater than $T$, but still the throughput of the system is greater than the throughput constraint $1/T$ when executed in a self-timed manner. This would be the case when the maximum cycle mean of the associated IPC graph is less than $T$. Also, the increase in the number of pipeline stages increases the number of delays in the system, which can be related to an increase in latency. Scheduling algorithms for Phase 1 that take these factors into account are expected to produce better results under self-timed execution.

The graph-theoretic framework that streamlines our scheduling strategies for deterministic, contention-free self-timed systems, is shown to be promising when it is applicable. The approach to “simulate” using graph-theoretic length computations executes much faster than usual event-driven simulation, and is therefore very useful for implementation aspects of schedule post-processing strategies described in the report.

In the later part of the report, the problem of how to change the configuration of a system where an application is executing on a polymorphous computing architecture, in accordance with a given set of performance requirements, is modeled and an approach to handle the PCA mapping problem for multiple applications and diverse performance requirements is presented. We have developed a simulation of this framework in conjunction with two simple, low-complexity approaches for power and performance estimation of a polymorphous embedded multiprocessing platform. The results demonstrate the ability of the framework to systematically adapt system configurations towards progressively better solutions for a variety of goals, even in the presence of significant uncertainties in application behavior.

Directions for future work in the PCA mapping problem include developing a model that can handle non-trivial metrics, such as the periodic-output latency metric defined in the initial part of the report, in a more natural way. Other approaches that can exploit the reconfigurability of the computing architecture in more flexible ways, such as by exploiting programmable message routing between processors, or by application of voltage scaling, would also be very useful.

## ACKNOWLEDGEMENTS

This research has been supported by the US National Science Foundation (9734275), and the Defense Advanced Research Projects Agency under Contract F30602-01-C-0171 through the University of Southern California Information Sciences Institute.

# REFERENCES

[1] A. Aiken and A. Nicolau. Optimal loop parallelization. In *Proceedings of the ACM Conference on Programming language Design and Implementation,* 1988.

[2] F. Baccelli, G. Cohen, G. J. Olsder, and J. Quadrat. *Synchronization and linearity.* John Wiley & Sons, Inc., 1992.

[3] S. Bakshi and D. D. Gajski, A scheduling and pipelining algorithm for Hardware/Software Systems, *Proceedings of the 10th International Symposium on Systems Synthesis (ISSS '97).*

[4] N. K. Bambha and S. S. Bhattacharyya, "A joint power/performance optimization technique for multiprocessor systems using a period graph construct", *Proceedings of the International Symposium on Systems Synthesis,* pages 91-97, Madrid, Spain, September 2000.

[5] S. Banerjee et al. Macro pipelining based scheduling on high performance heterogeneous multiprocessor systems. *IEEE transactions on signal processing,* June 1995.

[6] S. S. Bhattacharyya, P. K. Murthy, and E. A. Lee, S*oftware Synthesis from Dataflow Graphs*, Kluwer Academic Publishers, 1996.

[7] S. H. Bokhari, Partitioning problems in parallel, pipelined, and distributed computing. *IEEE Transactions on Computers*, Jan. 1988.

[8] R. K. Brayton and R. Spence, *Sensitivity and Optimization*, Elsevier Scientific Publishing Company, 1980.

[9] J. R. Budenske, R. S. Ramanujan, and H. J. Siegel, "On-Line Use of Off-Line Derived Mappings for Iterative Automatic Target Recognition Tasks and a Particular Class of Hardware Platforms", *6th Heterogeneous Computing Workshop (HCW'97),* co-sponsors: IEEE Computer Society and Office of Naval Research, pp. 96-110, Geneva, Switzerland, Apr. 1997.

[10] G. C. Buttazza and M. Caccamo, Minimizing aperiodic response times in a Firm Real-time Environment, *IEEE Transactions on Software Engineering,* Vol 25, No 1, Jan/Feb 1999, pp. 22-32.

[11] K. C. Cain, J. A. Torres, and R. T. Williams. *RT_STAP: Real-time space-time-adaptive processing benchmark.* Technical Report MTR 96B0000021, The MITRECorporation, February 1997.

[12] M. Charikar and S. Guha. improved combinatorial algorithms for facility location and k-median problems. *Proc. 40th Annual Symposium on Foundations of Computer Science*, 378-388, 1999.

[13] A. Choudhry et al. Optimal processor assignment for a class of pipelined computations, *IEEE transactions on parallel and distributed systems.* April 1994.

[14] A. Choudhry et al. Optimal processor assignment for a class of pipelined computations, *IEEE Transactions on parallel and distributed systems.* April 1994.

[15] F. Chudak, "Improved approximation algorithms for uncapaciateted facility location", In R. E. Bixby, E. A. Boyd and R. Z. Rios-Mercado, eds., *Integer Programming and Combinatorial Optimization*, Springer LNCS Vol. 1412, 180-194, 1998.

[16] T. Cormen et al. *Introduction to Algorithms*, McGraw Hill, 2000.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and company, 1999.

[18] S. Goddard, *On the management of latency in the synthesis of Real-time Signal processing systems from processing graphs,* Ph.D. Dissertation, University of North Carolina at Chapel Hill, Department of Computer Science, 1998.

[19] P. Hoang, *Compiling real time digital signal processing applications onto multiprocessor systems,* Ph.D. thesis.University of California at berkeley, June 1992.

[20] K. Jain and V. V. Vazirani, "Approximation algorithms for metric Facility location and k-median problems using the primal-dual scheme and Lagrangian relaxation", *Proc. Foundations of Computer Science, 1999.*

[21] B. K. Kim, Control latency for task assignment and scheduling of multiprocessor real-time control systems, *International Journal of Systems Science*, vol. 30, no. 1, pp. 123-130, Jan. 1999.

[22] K. Konstantinides et al., Task allocation and Scheduling Models for Multiprocessor Digital Signal Processing, *IEEE transactions on Acoustics, Speech, and Signal Processing*, Vol 38, No.12, Dec 1990, pp. 2151-2161.

[23] E. A. Lee, “Consistency in dataflow graphs”, *IEEE Transactions on Parallel and Distributed Systems*, April, 1991.

[24] C. Leiserson and J. Saxe, Retiming synchronous circuitry. *Algorithmica,* 6:5-35, 1991.

[25] V. Madisetti, *VLSI Digital Signal Processors.* IEEE Press, 1995.

[26] C. L. McCreary, A. A. Kahn, J. J. Thompson and M. E. McArdle, “A Comparison of Heuristics for Scheduling DAGS on Multiprocessors”, *International Parallel Processing Symposium*, 1994.

[27] G. D. Micheli, *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, 1994.

[28] M. D Natale, J. A. Stankovic, “Scheduling distributed real-time tasks with minimum jitter”, *IEEE Transactions on Computers*, Volume 49, Issue: 4, April 2000, pp. 303 -316.

[29] A. Papoulis, “*Probability, Random variables, and Stochastic processes*”, McGraw-Hill, 1991.

[30] J. L. Peterson, *Petri Net Theory and Modeling of Systems*, Prentice-Hall Inc., Englewoods Cliffs, New Jersey, 1981.

[31] L. L. Peterson and B. S. Davie, *Computer networks: A Systems Appraoch*, Morgan Kaufmann, 1996.

[32] S. Rajsbaum, M. Sidi, On the Performance of Synchronized Programs in Distributed Networks with Random Processing Times and Transmission Delays, *IEEE Trans. on Parallel and Distributed Systems,* Vol. 5, No. 9, Sept. 1994, pp. 939-950.

[33] R. Reiter, Scheduling parallel computations. *Journal of the association for computing machinery*, October 1968.

[34] D. B. Shmoys, E. Tardos, and K. I. Aardal. Approximation algorithms for facility location problems. *Proc. 29th ACM Symp. on Theory of Computing*, 265-274, 1997.

[35] G. N. Srinivasa Prasanna, Compilation of Parallel Multimedia Computation-Extending Retiming Theory and Amdahl’s Law, *ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, 1997, pp 180-192.

[36] S. Sriram, “Minimizing Communication and Synchronization Overhead in Multiprocessors for Digital Signal Processing,” Ph.D. Thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1995.

[37] S. Sriram and Shuvra S. Bhattacharyya, *Embedded Multiprocessors:Scheduling and Synchronization,* Marcel Dekker, 2000.

[38] J. Subhlok and G. Vondron, Optimal latency-throughput trade-offs for data parallel pipelines, *Proceedings of SPAA’ 96,* June 96.

[39] F. M. Tsou et al., “Design and simulation of an efficient real-time traffic scheduler with jitter and delay guarantees”, *IEEE Transactions on Multimedia,* Volume 2 Issue 4, Dec. 2000, pp. 255-266.

[40] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe and A. Agarwal, “Baring it all to Software: Raw Machines”, *IEEE Computer,* September 1997, pp. 86-93.

[41] Ti-Yen and Wayne Wolf, Performance estimation for Real-Time distributed embedded systems, *IEEE Transactions on parallel and distributed systems,* vol. 9, No.11, Nov1998, pp.1125-1136.

[42] A. Y. Zomaya, “Parallel and Distributed Computing: The Scene, the Props, the Players,” *Parallel and Distributed Computing Handbook,* A.Y. Zomaya, ed., pp. 5-23, New York: McGraw-Hill, 1996.